

BOOK 5: TELEGRAM BOT INTEGRATION

Complete Guide to Building Crypto Bots on Telegram (2025 Edition)

INTRODUCTION

Telegram is no longer just a messaging app. In 2025, it has become a complete operating system for crypto with over one billion monthly active users, a built-in blockchain (TON), native crypto wallets for 100+ million users, and Mini Apps that function like full applications without leaving the chat. This book teaches you to build within this ecosystem.

The transformation happened fast. In early 2024, Telegram had bots. By late 2025, Telegram has an entire economy. Telegram Stars power digital payments. TON blockchain is the exclusive infrastructure for Mini Apps. The TON Wallet serves 87 million Americans and hundreds of millions globally. Trading bots like Trojan (formerly Unibot), Maestro, and Banana Gun process billions in volume monthly.

This isn't theory. This is where crypto actually happens now.

WHY TELEGRAM DOMINATES CRYPTO IN 2025

The numbers tell the story:

One billion monthly active users. Not accounts. Active users. This is the largest captive audience for any messaging platform. When you build on Telegram, you're building for a billion people.

100+ million activated crypto wallets. The TON Wallet integration means your bot doesn't need to teach users about wallets. They already have one. Self-custodial. Built in. Ready to transact.

Telegram Stars processed hundreds of millions in digital goods. The mandatory payment system for Mini Apps created a real economy. Users buy, creators earn, everyone builds.

Trading bots moved over 50 billion dollars in 2025. Trojan alone did 24 billion. Maestro did 13 billion. Banana Gun did 12 billion. These aren't side projects - they're financial infrastructure.

Mini Apps 2.0 launched with full-screen mode, device sensors, secure storage, and subscription payments. What started as simple web views became genuine applications.

THE TON EXCLUSIVITY SHIFT

In January 2025, everything changed. TON Foundation announced exclusive partnership with Telegram. What this means:

TON is the only blockchain for Telegram Mini Apps. Period. If your Mini App uses blockchain, it uses TON. Developers had until February 21, 2025 to migrate from Ethereum, Solana, or any other chain. The ecosystem consolidated.

TON Connect is the mandatory wallet protocol. No WalletConnect. No custom integrations. TON Connect or nothing. This standardization actually helps developers - one protocol to learn, one integration to maintain.

All crypto payments in Telegram use Toncoin. Telegram Stars, Telegram Premium, Telegram Ads, developer payouts - everything flows through TON. This creates genuine demand for the token and genuine utility for the blockchain.

NFTs for Telegram assets live on TON. Emojis, stickers, limited edition gifts - all tokenized on TON blockchain. Digital ownership native to the platform.

This isn't optional. If you're building crypto on Telegram in 2025, you're building on TON.

WHAT THIS BOOK COVERS

Chapter 1: Telegram Ecosystem 2025

The complete landscape. Bot API 8.x features including message streaming and Business 2.0. Mini Apps 2.0 capabilities. TON integration architecture. Telegram Stars economy. Understanding where everything fits before building anything.

Chapter 2: Bot Development Fundamentals

Building bots with python-telegram-bot v22 and aiogram. Async patterns that scale. Command handlers, conversation flows, inline keyboards. Webhooks versus polling. Error handling and resilience. The foundation everything else builds on.

Chapter 3: Mini Apps Development

Full-screen applications within Telegram. Device sensors for gaming. DeviceStorage and SecureStorage for state. Web3 integration patterns. Subscription models with Telegram Stars. Building apps that feel native.

Chapter 4: Telegram Stars and Payments

The complete payments infrastructure. Creating invoices for digital goods. Subscription billing with automatic renewal. Withdrawing Stars to TON via Fragment. Compliance with app store rules. Monetization strategies that actually work.

Chapter 5: TON Blockchain Integration

Deep TON development. TON Connect wallet integration. Smart contract interaction from bots. Jettons (TON tokens) and NFTs. Transaction monitoring and confirmation. Building on TON's architecture.

Chapter 6: Trading Bot Architecture

How the billion-dollar bots work. Token sniping with anti-rug protection. Copy trading wallet monitoring. MEV resistance patterns. Multi-chain coordination (within TON ecosystem). Risk management and position sizing.

Chapter 7: Community Management

Automating groups at scale. Token-gated access using NFT ownership. Anti-spam and anti-scam protection. Engagement tools and gamification. Whale alerts and market notifications. Managing

200,000 member communities.

Chapter 8: Production Deployment

Taking bots from development to billions in volume. Webhook scaling for high throughput. Database design for bot state.

Monitoring millions of transactions. Security for systems handling real money. Operational excellence.

THE ARCHITECTURE OF TELEGRAM CRYPTO

Understanding the layers helps you build correctly:

Layer 1 - Telegram Platform:

The messaging infrastructure. Bots, groups, channels, Mini Apps. Bot API for automation. Web Apps API for Mini Apps. This is the interface users see and interact with.

Layer 2 - TON Blockchain:

The value layer. TON Wallet for user funds. Toncoin for transactions. TON Connect for wallet integration. Smart contracts for logic. This is where money moves and ownership lives.

Layer 3 - Your Application:

Your bot, Mini App, or trading system. Uses Layer 1 for user interface. Uses Layer 2 for value transfer. Adds business logic, data storage, and integration with external systems.

Layer 4 - External Systems:

Price feeds, exchange APIs, analytics services, external databases. Your application connects Telegram users to the broader crypto ecosystem.

Every feature you build touches multiple layers. A trading bot receives commands via Bot API (Layer 1), executes swaps on TON DEXs (Layer 2), implements your trading logic (Layer 3), and fetches prices from aggregators (Layer 4).

TELEGRAM BOT API 8.X HIGHLIGHTS

The Bot API evolved significantly in 2025:

Message Streaming allows bots to stream partial messages while generating content. The `sendMessageDraft` method lets AI-powered bots show thinking in real-time. Users see responses form instead of waiting for completion.

Business 2.0 lets bots manage business accounts completely. Customize name, username, bio, profile pictures. Handle customer messages. Mark conversations read. Full CRM integration through Telegram.

Telegram Stars Integration is native to the Bot API. Create invoices, process payments, manage subscriptions - all through standard bot methods. No external payment processors for digital goods.

Emoji Status Management through `setUserEmojiStatus`. Mini Apps can prompt users to set emoji status, enabling viral distribution and status signaling.

Prepared Inline Messages via `savePreparedInlineMessage`. Share media from Mini Apps to any chat or post to stories. Referral codes, memes, artwork - shareable content creation.

Transaction Support for Chats. Bots can now handle transactions at the chat level, not just user level. Group treasuries become possible.

User ID Changes: After upcoming updates, user IDs will exceed 2^{31} . Your code must handle 64-bit integers. This seems minor until it breaks your database.

MINI APPS 2.0 CAPABILITIES

Mini Apps became real applications:

Full-Screen Mode in portrait and landscape. `requestFullscreen` and `exitFullscreen` methods. Games, video players, immersive experiences - no longer constrained to partial screen.

Device Sensors including Accelerometer, DeviceOrientation, and Gyroscope. Motion-controlled games. VR experiences. Fitness tracking. The phone's sensors are available to your code.

Safe Area Management through safeAreaInset and contentSafeAreaInset. Your UI respects notches, rounded corners, and system elements. Professional apps, not awkward overlaps.

Storage Options with DeviceStorage for persistent local data and SecureStorage for sensitive information. Offline capability. Secure credential storage. Real application architecture.

Orientation Control via lockOrientation and unlockOrientation. Games that require landscape. Apps that require portrait. You control the experience.

Media Sharing to any chat or stories. Users can share content your Mini App creates. Viral distribution built into the platform.

Subscription Payments with Telegram Stars. Recurring billing, multiple tiers, automatic renewal. Monetization infrastructure provided by the platform.

THE TRADING BOT ECOSYSTEM

Understanding the competitive landscape:

Trojan (formerly Unibot) leads with 2 million users and 24+ billion in volume. Multi-chain support across Ethereum, Arbitrum, Base, and Solana. AI-driven algorithms. Revenue sharing with UNIBOT token holders. The scale proves the market.

Maestro serves 600,000 users across 10+ chains with 13+ billion in volume. Token sniping with anti-rug detection. Whale tracking via Whale Bot. Flexible pricing with free tier (1% tax) or Premium (\$200/month).

Banana Gun matches Maestro's user base with 12+ billion in

volume on Solana, Base, and Blast. Auto sniping, copy trading, MEV protection. 40% of fees distributed to BANANA holders.

BONKbot, GMGN AI, and dozens of others compete for niches. The market is large enough for specialization.

The pattern is clear: these bots handle real money, real volume, real users. Building in this space means building financial infrastructure.

PREREQUISITES FOR THIS BOOK

This book assumes prior knowledge:

Books 1-4 of this series or equivalent. You need blockchain fundamentals, smart contract interaction, and infrastructure patterns. We reference those concepts constantly.

Python at intermediate level. `async/await`, classes, error handling. The examples use `python-telegram-bot v22` and `aiogram`. JavaScript knowledge helps for Mini Apps.

Web development basics. HTML, CSS, JavaScript for Mini Apps. React or Vue experience useful. API design understanding.

Database fundamentals. PostgreSQL for bot state. Redis for caching. Schema design and query optimization.

New to this book - TON development. We teach TON-specific concepts. FunC smart contract language basics. TON Virtual Machine understanding. Jettons and NFT standards.

A NOTE ON SECURITY AND RESPONSIBILITY

The stakes are higher than ever:

Trading bots handle billions in volume. A bug doesn't just cause user frustration - it causes financial loss. Security isn't a feature, it's the foundation.

Private keys in Telegram pose risks. Users trust your bot with wallet access. That trust, broken once, destroys reputation permanently. Design for the attack you haven't imagined.

Regulatory attention grows. Telegram's crypto features draw regulator interest. Build compliant systems. Document thoroughly. Know your jurisdictions.

The bots in this book handle real money. Test like production will attack you. Build like your reputation depends on it - because it does.

Let's build the next billion-dollar bot.

SOURCES AND REFERENCES

Official Telegram Documentation:

- Bot API: core.telegram.org/bots/api
- Bot API Changelog: core.telegram.org/bots/api-changelog
- Mini Apps: core.telegram.org/bots/webapps
- Payments with Stars: core.telegram.org/bots/payments-stars

TON Documentation:

- TON Docs: docs.ton.org
- TON Connect: docs.ton.org/v3/guidelines/ton-connect
- TON Connect SDK: github.com/ton-connect/sdk

Libraries:

- python-telegram-bot: docs.python-telegram-bot.org
- aiogram: github.com/aiogram/aiogram
- @tonconnect/sdk: npmjs.com/package/@tonconnect/sdk

News and Updates:

- @BotNews on Telegram
- @taboraofficial on Telegram (TON)
- blog.ton.org

CHAPTER 1: TELEGRAM BOT FUNDAMENTALS

Understanding the Bot API from first principles. This chapter builds your foundation - how Telegram bots actually work, how to create them, and how to handle user interactions reliably. Everything in later chapters depends on mastering these fundamentals.

SECTION 1: HOW TELEGRAM BOTS WORK

Telegram bots are special accounts operated by software rather than humans. They can receive messages, send responses, create interactive keyboards, and integrate with external services. But they're not magic - they follow specific rules and limitations you must understand.

The Bot API is HTTP-based. Your code makes HTTPS requests to Telegram's servers. Send a message? POST request. Get updates? GET request. Everything flows through `api.telegram.org` with your bot token authenticating each request.

Bots cannot initiate conversations. A user must start the interaction - either by sending a message, clicking a deep link, or being added to a group. This isn't a bug, it's spam prevention. Your bot responds to users, never cold-messages them.

Bot tokens are secrets. The token format looks like `123456789:ABCdefGHIjklMNOpqrsTUVwxyz`. The first part is your bot's user ID. The second part is the authentication key. Anyone with your token controls your bot completely. Treat it like a private key.

Creating a bot starts with BotFather. This is Telegram's official bot for managing bots. Send `/newbot`, answer the prompts, receive your token. The process takes thirty seconds.

This Python code demonstrates the raw API without libraries:

```
import requests
import json
```

```
class RawTelegramAPI:
```

```

def __init__(self, token):
    self.token = token
    self.base_url = f"https://api.telegram.org/bot{token}"

    def make_request(self, method, params = None):
        url = f"{self.base_url}/{method}"
        response = requests.post(url, json = params or {})
        result = response.json()

        if not result.get("ok"):
            raise Exception(f"API Error: {result.get('description')}")

        return result.get("result")

    def get_me(self):
        return self.make_request("getMe")

    def send_message(self, chat_id, text, reply_markup = None):
        params = {
            "chat_id": chat_id,
            "text": text,
            "parse_mode": "HTML"
        }

        if reply_markup:
            params["reply_markup"] = reply_markup

        return self.make_request("sendMessage", params)

    def get_updates(self, offset = None, timeout = 30):
        params = {"timeout": timeout}

        if offset:
            params["offset"] = offset

        return self.make_request("getUpdates", params)

api = RawTelegramAPI("YOUR_BOT_TOKEN")

```

```
bot_info = api.get_me()
print(f'Bot: @{bot_info["username"]}')

api.send_message(
    chat_id = 123456789,
    text = "Hello from raw API!"
)
```

The raw API shows what's happening under the hood. Every bot library ultimately makes these same HTTP requests. Understanding this layer helps debug issues when library abstractions hide the problem.

SECTION 2: POLLING VERSUS WEBHOOKS

Two methods exist for receiving updates from Telegram. Polling asks Telegram repeatedly "any new messages?" Webhooks tell Telegram "send new messages to this URL." Both work. The right choice depends on your situation.

Polling is simpler for development. Your code runs anywhere - local machine, server without public IP, behind corporate firewalls. No SSL certificates needed. No domain required. Just run the script and it works.

The polling loop continuously fetches updates:

```
import asyncio
from telegram import Update
from telegram.ext import Application, CommandHandler,
MessageHandler, filters

class PollingBot:

    def __init__(self, token):
        self.application = Application.builder().token(token).build()
        self.setup_handlers()

    def setup_handlers(self):
```

```

self.application.add_handler(
    CommandHandler("start", self.handle_start)
)
self.application.add_handler(
    CommandHandler("help", self.handle_help)
)
self.application.add_handler(
    MessageHandler(filters.TEXT & ~filters.COMMAND,
self.handle_message)
)

```

```

async def handle_start(self, update, context):
    user = update.effective_user
    await update.message.reply_text(
        f'Welcome {user.first_name}! I'm your crypto assistant.\n\n'
        f'Use /help to see available commands.'
    )

```

```

async def handle_help(self, update, context):
    help_text = (
        "<b> Available Commands </b> \n\n"
        "/start - Start the bot\n"
        "/help - Show this message\n"
        "/price [symbol] - Get token price\n"
        "/wallet - View your wallet"
    )
    await update.message.reply_text(help_text, parse_mode="HTML")

```

```

async def handle_message(self, update, context):
    text = update.message.text
    await update.message.reply_text(f'You said: {text}')

```

```

def run(self):
    self.application.run_polling(
        allowed_updates=Update.ALL_TYPES,
        drop_pending_updates=True
    )

```

```

bot = PollingBot("YOUR_TOKEN")
bot.run()

```

Webhooks are better for production. No constant polling means lower latency. Updates arrive instantly via HTTP POST to your server. More efficient for high-volume bots. Required for serverless deployments.

Setting up webhooks requires HTTPS. Telegram only sends webhooks to secure endpoints. You need a domain, SSL certificate, and publicly accessible server.

```
from flask import Flask, request
from telegram import Update, Bot
from telegram.ext import Application, CommandHandler
import asyncio

app = Flask(__name__)

class WebhookBot:

    def __init__(self, token, webhook_url):
        self.token = token
        self.webhook_url = webhook_url
        self.bot = Bot(token)
        self.application = Application.builder().token(token).build()
        self.setup_handlers()

    def setup_handlers(self):
        self.application.add_handler(
            CommandHandler("start", self.handle_start)
        )
        self.application.add_handler(
            CommandHandler("price", self.handle_price)
        )

    async def handle_start(self, update, context):
        await update.message.reply_text("Webhook bot is running!")

    async def handle_price(self, update, context):
        args = context.args
        if not args:
```

```
await update.message.reply_text("Usage: /price ETH")
return
```

```
symbol = args[0].upper()
await update.message.reply_text(f'Fetching price for {symbol}...')
```

```
async def set_webhook(self):
    await self.bot.set_webhook(
        url=f'{self.webhook_url}/webhook/{self.token}',
        allowed_updates=["message", "callback_query"],
        drop_pending_updates=True
    )
```

```
async def process_update(self, update_data):
    update = Update.de_json(update_data, self.bot)
    await self.application.process_update(update)
```

```
def get_flask_app(self):
    return app
```

```
webhook_bot = None
```

```
def create_bot(token, webhook_url):
    global webhook_bot
    webhook_bot = WebhookBot(token, webhook_url)
    return webhook_bot
```

```
@app.route(f"/webhook/<token>", methods=["POST"])
def webhook_handler(token):
    if webhook_bot and token == webhook_bot.token:
        update_data = request.get_json()
        asyncio.run(webhook_bot.process_update(update_data))
    return "OK", 200
    return "Unauthorized", 401
```

```
@app.route("/health")
def health():
    return "OK", 200
```

The hybrid approach works well during development. Run polling

locally for testing, deploy webhooks for production. Same handler code, different update delivery.

SECTION 3: HANDLING MESSAGES AND COMMANDS

Messages come in many forms. Text, photos, documents, locations, contacts, stickers. Your bot must handle what it supports and gracefully ignore what it doesn't.

Commands follow a pattern: /command or /command@botusername. The @ suffix appears in groups to direct commands to specific bots. Your handlers should work with or without it.

This comprehensive message handler demonstrates the patterns:

```
from telegram import Update
from telegram.ext import (
    Application, CommandHandler, MessageHandler,
    filters, ContextTypes
)
from datetime import datetime

class MessageBot:

    def __init__(self, token):
        self.application = Application.builder().token(token).build()
        self.user_data = {}
        self.setup_handlers()

    def setup_handlers(self):
        self.application.add_handler(
            CommandHandler("start", self.cmd_start)
        )
        self.application.add_handler(
            CommandHandler("balance", self.cmd_balance)
        )
        self.application.add_handler(
            CommandHandler("send", self.cmd_send)
```

```

)

self.application.add_handler(
    MessageHandler(filters.TEXT & ~filters.COMMAND, self.on_text)
)
self.application.add_handler(
    MessageHandler(filters.PHOTO, self.on_photo)
)
self.application.add_handler(
    MessageHandler(filters.Document.ALL, self.on_document)
)

self.application.add_error_handler(self.on_error)

async def cmd_start(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
    user = update.effective_user
    chat_id = update.effective_chat.id

    self.user_data[user.id] = {
        "username": user.username,
        "first_name": user.first_name,
        "chat_id": chat_id,
        "joined": datetime.utcnow().isoformat(),
        "balance": 0.0
    }

    welcome = (
        f"Welcome to CryptoBot, {user.first_name}!\n\n"
        f"Your user ID: <code>{user.id}</code> \n"
        f"Your chat ID: <code>{chat_id}</code> \n\n"
        f"Commands:\n"
        f"/balance - Check your balance\n"
        f"/send [amount] [address] - Send crypto"
    )

    await update.message.reply_text(welcome, parse_mode="HTML")

    async def cmd_balance(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):

```



```
user_id = update.effective_user.id
```

```
if user_id not in self.user_data:  
    await update.message.reply_text(  
        "Please /start first to register."  
    )  
    return
```

```
balance = self.user_data[user_id].get("balance", 0)
```

```
await update.message.reply_text(  
    f'Your balance: <b>{balance:.6f}</b> ETH</b>',  
    parse_mode="HTML"  
)
```

```
async def cmd_send(self, update: Update, context:  
    ContextTypes.DEFAULT_TYPE):  
    args = context.args
```

```
if len(args) < 2:  
    await update.message.reply_text(  
        "Usage: /send [amount] [address]\n"  
        "Example: /send 0.1 0x742d35Cc..."  
    )  
    return
```

```
try:  
    amount = float(args[0])  
except ValueError:  
    await update.message.reply_text("Invalid amount. Use a number.")  
    return
```

```
address = args[1]
```

```
if not address.startswith("0x") or len(address) != 42:  
    await update.message.reply_text("Invalid Ethereum address.")  
    return
```

```
user_id = update.effective_user.id  
balance = self.user_data.get(user_id, {}).get("balance", 0)
```

```
if amount > balance:
    await update.message.reply_text(
        f'Insufficient balance. You have {balance:.6f} ETH.'
    )
return
```

```
await update.message.reply_text(
    f'Sending {amount} ETH to {address[:10]}...{address[-6:]} \n'
    f'Please confirm this transaction.',
    parse_mode="HTML"
)
```

```
async def on_text(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
    text = update.message.text.lower()
```

```
if "price" in text:
    await update.message.reply_text(
        "To check prices, use /price [symbol] \n"
        "Example: /price ETH"
    )
elif "help" in text:
    await update.message.reply_text(
        "Use /start to see available commands."
    )
else:
    await update.message.reply_text(
        "I don't understand that. Use /help for commands."
    )
```

```
async def on_photo(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
    photo = update.message.photo[-1]
    file_id = photo.file_id
```

```
await update.message.reply_text(
    f'Received photo! \n'
    f'File ID: <code>{file_id[:20]}...</code> \n'
    f'Size: {photo.width}x{photo.height}',
```

```

parse_mode = "HTML"
)

async def on_document(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
    doc = update.message.document

    await update.message.reply_text(
        f'Received document: {doc.file_name}\n'
        f'Size: {doc.file_size} bytes\n'
        f'Type: {doc.mime_type}'
    )

    async def on_error(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
        print(f'Error: {context.error}')

        if update and update.effective_message:
            await update.effective_message.reply_text(
                "An error occurred. Please try again."
            )

    def run(self):
        self.application.run_polling()

```

SECTION 4: INLINE KEYBOARDS AND CALLBACKS

Inline keyboards transform chat interfaces. Instead of typing commands, users tap buttons. The buttons can trigger callbacks, open URLs, or switch to inline mode. This creates app-like experiences within Telegram.

Callback queries are the key mechanism. When a user taps an inline button, Telegram sends a callback query to your bot. The query contains data you attached to the button. Your bot processes the data and responds.

```

from telegram import Update, InlineKeyboardButton,
InlineKeyboardMarkup

```

```
from telegram.ext import (
    Application, CommandHandler, CallbackQueryHandler,
    ContextTypes
)
```

```
class KeyboardBot:
```

```
    def __init__(self, token):
        self.application = Application.builder().token(token).build()
        self.setup_handlers()
```

```
    def setup_handlers(self):
        self.application.add_handler(
            CommandHandler("menu", self.show_menu)
        )
        self.application.add_handler(
            CommandHandler("trade", self.show_trade_options)
        )
        self.application.add_handler(
            CallbackQueryHandler(self.handle_callback)
        )
```

```
    async def show_menu(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
        keyboard = [
            [
                InlineKeyboardButton("Wallet", callback_data="menu:wallet"),
                InlineKeyboardButton("Trade", callback_data="menu:trade")
            ],
            [
                InlineKeyboardButton("Prices", callback_data="menu:prices"),
                InlineKeyboardButton("Settings", callback_data="menu:settings")
            ],
            [
                InlineKeyboardButton("Help", callback_data="menu:help")
            ]
        ]
```

```
        reply_markup = InlineKeyboardMarkup(keyboard)
```

```
await update.message.reply_text(
    "What would you like to do?",
    reply_markup=reply_markup
)
```

```
async def show_trade_options(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
    keyboard = [
        [
            InlineKeyboardButton("Buy ETH", callback_data="trade:buy:ETH"),
            InlineKeyboardButton("Sell ETH", callback_data="trade:sell:ETH")
        ],
        [
            InlineKeyboardButton("Buy BTC", callback_data="trade:buy:BTC"),
            InlineKeyboardButton("Sell BTC", callback_data="trade:sell:BTC")
        ],
        [
            InlineKeyboardButton("Custom Trade",
callback_data="trade:custom"),
            InlineKeyboardButton("Back", callback_data="menu:main")
        ]
    ]
```

```
reply_markup = InlineKeyboardMarkup(keyboard)
```

```
message = update.message or update.callback_query.message
```

```
if update.callback_query:
    await message.edit_text(
        "Select a trading pair:",
        reply_markup=reply_markup
    )
else:
    await message.reply_text(
        "Select a trading pair:",
        reply_markup=reply_markup
    )
```

```
async def handle_callback(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
```

```
query = update.callback_query
await query.answer()
```

```
data = query.data
parts = data.split(":")
```

```
if parts[0] == "menu":
    await self.handle_menu_callback(query, parts[1])
elif parts[0] == "trade":
    await self.handle_trade_callback(query, parts[1:])
elif parts[0] == "confirm":
    await self.handle_confirm_callback(query, parts[1:])
```

```
async def handle_menu_callback(self, query, action):
    if action == "wallet":
        await query.message.edit_text(
            "Your Wallet\n\n"
            "ETH: 1.5432\n"
            "BTC: 0.0521\n"
            "USDT: 1,234.56",
            reply_markup=self.back_button("menu:main")
        )
```

```
    elif action == "trade":
        await self.show_trade_options(
            Update(update_id=0, callback_query=query),
            None
        )
```

```
    elif action == "prices":
        await query.message.edit_text(
            "Current Prices\n\n"
            "ETH: $2,345.67\n"
            "BTC: $43,210.00\n"
            "SOL: $98.76",
            reply_markup=self.back_button("menu:main")
        )
```

```
    elif action == "settings":
        keyboard = [
```

```

[
InlineKeyboardButton(
"Notifications: ON",
callback_data = "settings:notifications:toggle"
)
],
[
InlineKeyboardButton(
"Currency: USD",
callback_data = "settings:currency:select"
)
],
[
InlineKeyboardButton("Back", callback_data = "menu:main")
]
]
await query.message.edit_text(
"Settings",
reply_markup = InlineKeyboardMarkup(keyboard)
)

elif action == "main":
keyboard = [
[
InlineKeyboardButton("Wallet", callback_data = "menu:wallet"),
InlineKeyboardButton("Trade", callback_data = "menu:trade")
],
[
InlineKeyboardButton("Prices", callback_data = "menu:prices"),
InlineKeyboardButton("Settings", callback_data = "menu:settings")
]
]
await query.message.edit_text(
"What would you like to do?",
reply_markup = InlineKeyboardMarkup(keyboard)
)

async def handle_trade_callback(self, query, parts):
if parts[0] == "buy" or parts[0] == "sell":
action = parts[0].upper()

```

```
symbol = parts[1]
```

```
keyboard = [  
    [  
        InlineKeyboardButton("0.1", callback_data = f'confirm:{parts[0]}:  
{symbol}:0.1"),  
        InlineKeyboardButton("0.5", callback_data = f'confirm:{parts[0]}:  
{symbol}:0.5"),  
        InlineKeyboardButton("1.0", callback_data = f'confirm:{parts[0]}:  
{symbol}:1.0")  
    ],  
    [  
        InlineKeyboardButton("Custom Amount",  
callback_data = f'trade:amount:{parts[0]}:{symbol}')  
    ],  
    [  
        InlineKeyboardButton("Back", callback_data = "menu:trade")  
    ]  
]
```

```
await query.message.edit_text(  
    f'{action} {symbol}\n\n"  
    f'Current price: $2,345.67\n"  
    f'Select amount:',  
    reply_markup = InlineKeyboardMarkup(keyboard)  
)
```

```
async def handle_confirm_callback(self, query, parts):  
    action = parts[0]  
    symbol = parts[1]  
    amount = parts[2]
```

```
keyboard = [  
    [  
        InlineKeyboardButton("Confirm", callback_data = f'execute:  
{action}:{symbol}:{amount}'),  
        InlineKeyboardButton("Cancel", callback_data = "menu:trade")  
    ]  
]
```



```

await query.message.edit_text(
f'Confirm Transaction\n\n'
f'Action: {action.upper()}\n'
f'Asset: {symbol}\n'
f'Amount: {amount}\n'
f'Price: $2,345.67\n'
f'Total: ${float(amount) * 2345.67:,.2f}',
reply_markup = InlineKeyboardMarkup(keyboard)
)

```

```

def back_button(self, callback_data):
return InlineKeyboardMarkup([
[InlineKeyboardButton("Back", callback_data = callback_data)]
])

```

```

def run(self):
self.application.run_polling()

```

SECTION 5: CONVERSATION HANDLERS

Some interactions require multiple steps. Setting up a wallet needs address input. Executing a trade needs confirmation. These multi-step flows use conversation handlers that track state across messages.

The conversation handler manages state transitions. Each state defines what input is expected and which state comes next. The conversation ends when reaching a terminal state or timing out.

```

from telegram import Update, InlineKeyboardButton,
InlineKeyboardMarkup
from telegram.ext import (
Application, CommandHandler, MessageHandler,
CallbackQueryHandler, ConversationHandler, filters,
ContextTypes
)

```

```

WAITING_AMOUNT = 1
WAITING_ADDRESS = 2

```

WAITING_CONFIRMATION = 3

class ConversationBot:

```
def __init__(self, token):
    self.application = Application.builder().token(token).build()
    self.pending_transactions = {}
    self.setup_handlers()
```

```
def setup_handlers(self):
    send_conversation = ConversationHandler(
        entry_points=[
            CommandHandler("send", self.send_start)
        ],
        states={
            WAITING_AMOUNT: [
                MessageHandler(filters.TEXT & ~filters.COMMAND,
                                self.send_amount)
            ],
            WAITING_ADDRESS: [
                MessageHandler(filters.TEXT & ~filters.COMMAND,
                                self.send_address)
            ],
            WAITING_CONFIRMATION: [
                CallbackQueryHandler(self.send_confirm, pattern="^send:")
            ]
        },
        fallbacks=[
            CommandHandler("cancel", self.cancel)
        ],
        conversation_timeout=300
    )
```

```
self.application.add_handler(send_conversation)
self.application.add_handler(CommandHandler("start",
self.cmd_start))
```

```
async def cmd_start(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
    await update.message.reply_text(
```

```
"Welcome! Use /send to transfer crypto."  
)
```

```
async def send_start(self, update: Update, context:  
ContextTypes.DEFAULT_TYPE):  
    context.user_data["send"] = {}
```

```
    await update.message.reply_text(  
        "Send Crypto\n\n"  
        "Enter the amount to send (in ETH):\n"  
        "Example: 0.5\n\n"  
        "Use /cancel to abort."  
    )
```

```
    return WAITING_AMOUNT
```

```
async def send_amount(self, update: Update, context:  
ContextTypes.DEFAULT_TYPE):  
    text = update.message.text.strip()
```

```
    try:  
        amount = float(text)  
    except ValueError:  
        await update.message.reply_text(  
            "Invalid amount. Please enter a number.\n"  
            "Example: 0.5"  
        )  
    return WAITING_AMOUNT
```

```
    if amount <= 0:  
        await update.message.reply_text(  
            "Amount must be greater than zero."  
        )  
    return WAITING_AMOUNT
```

```
    if amount > 10:  
        await update.message.reply_text(  
            "Maximum send amount is 10 ETH per transaction."  
        )  
    return WAITING_AMOUNT
```

```
context.user_data["send"]["amount"] = amount
```

```
await update.message.reply_text(
    f'Amount: {amount} ETH\n\n'
    f'Now enter the destination address:\n'
    f'Example: 0x742d35Cc6634C0532925a3b844Bc9e7595f...'
)
```

```
return WAITING_ADDRESS
```

```
async def send_address(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
    address = update.message.text.strip()
```

```
if not address.startswith("0x"):
    await update.message.reply_text(
        "Address must start with 0x"
    )
return WAITING_ADDRESS
```

```
if len(address) != 42:
    await update.message.reply_text(
        "Invalid address length. Ethereum addresses are 42 characters."
    )
return WAITING_ADDRESS
```

```
try:
    int(address, 16)
except ValueError:
    await update.message.reply_text(
        "Address contains invalid characters."
    )
return WAITING_ADDRESS
```

```
context.user_data["send"]["address"] = address
amount = context.user_data["send"]["amount"]
```

```
keyboard = [
    [
```

```
InlineKeyboardButton("Confirm", callback_data="send:confirm"),
InlineKeyboardButton("Cancel", callback_data="send:cancel")
]
]
```

```
short_address = f'{address[:10]}...{address[-8:]}'
```

```
await update.message.reply_text(
f"Transaction Summary\n\n"
f"Amount: {amount} ETH\n"
f"To: {short_address}\n"
f"Fee: ~0.002 ETH\n"
f"Total: {amount + 0.002} ETH\n\n"
f"Confirm this transaction?",
reply_markup=InlineKeyboardMarkup(keyboard)
)
```

```
return WAITING_CONFIRMATION
```

```
async def send_confirm(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
query = update.callback_query
await query.answer()
```

```
action = query.data.split(":")[1]
```

```
if action == "cancel":
await query.message.edit_text(
"Transaction cancelled."
)
)
```

```
return ConversationHandler.END
```

```
send_data = context.user_data.get("send", {})
amount = send_data.get("amount")
address = send_data.get("address")
```

```
await query.message.edit_text(
"Processing transaction..."
)
)
```

```
tx_hash = "0x" + "a" * 64
```

```
await query.message.edit_text(
    f"Transaction Sent!\n\n"
    f"Amount: {amount} ETH\n"
    f"To: {address[:10]}...{address[-8:]}\n"
    f"TX: {tx_hash[:20]}...\n\n"
    f"View on Etherscan"
)
```

```
context.user_data["send"] = {}
```

```
return ConversationHandler.END
```

```
async def cancel(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
    context.user_data["send"] = {}
```

```
await update.message.reply_text(
    "Operation cancelled."
)
```

```
return ConversationHandler.END
```

```
def run(self):
    self.application.run_polling()
```

SECTION 6: GROUP AND CHANNEL HANDLING

Bots behave differently in groups than in private chats. Group messages may not be received unless privacy mode is disabled. Commands need @botusername to address specific bots. Admin permissions control what bots can do.

Understanding chat types prevents confusion:

```
from telegram import Update, ChatMember
from telegram.ext import (
    Application, CommandHandler, MessageHandler,
```

```
ChatMemberHandler, filters, ContextTypes
)
```

```
class GroupBot:
```

```
    def __init__(self, token):
        self.application = Application.builder().token(token).build()
        self.group_settings = {}
        self.setup_handlers()
```

```
    def setup_handlers(self):
        self.application.add_handler(
            CommandHandler("start", self.cmd_start)
        )
        self.application.add_handler(
            CommandHandler("setup", self.cmd_setup)
        )
        self.application.add_handler(
            CommandHandler("stats", self.cmd_stats)
        )
        self.application.add_handler(
            ChatMemberHandler(self.on_chat_member,
            ChatMemberHandler.MY_CHAT_MEMBER)
        )
        self.application.add_handler(
            MessageHandler(filters.StatusUpdate.NEW_CHAT_MEMBERS,
            self.on_new_member)
        )
```

```
    async def cmd_start(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
        chat = update.effective_chat
```

```
        if chat.type == "private":
            await update.message.reply_text(
                "Welcome! Add me to a group to use my features.\n\n"
                "In groups, I can:\n"
                "- Track crypto prices\n"
                "- Provide market updates\n"
                "- Run community games"
```

```
)  
else:  
    await update.message.reply_text(  
        f'Bot active in {chat.title}!\n"  
        f'Use /setup to configure settings."  
    )
```

```
async def cmd_setup(self, update: Update, context:  
ContextTypes.DEFAULT_TYPE):  
    chat = update.effective_chat  
    user = update.effective_user
```

```
    if chat.type == "private":  
        await update.message.reply_text(  
            "This command only works in groups."  
        )  
    return
```

```
    member = await chat.get_member(user.id)
```

```
    if member.status not in ["administrator", "creator"]:  
        await update.message.reply_text(  
            "Only admins can configure the bot."  
        )  
    return
```

```
    chat_id = chat.id
```

```
    self.group_settings[chat_id] = {  
        "price_alerts": True,  
        "welcome_message": True,  
        "language": "en"  
    }
```

```
    await update.message.reply_text(  
        f'Group configured!\n\n"  
        f'Chat ID: {chat_id}\n"  
        f'Chat Type: {chat.type}\n"  
        f'Member Count: {await chat.get_member_count()}"  
    )
```



```

async def cmd_stats(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
    chat = update.effective_chat

    if chat.type == "private":
        await update.message.reply_text(
            f'Your user ID: {update.effective_user.id}'
        )
    return

    member_count = await chat.get_member_count()
    admin_count = len(await chat.get_administrators())

    await update.message.reply_text(
        f'Group Statistics\n\n'
        f'Name: {chat.title}\n'
        f'Type: {chat.type}\n'
        f'Members: {member_count}\n'
        f'Admins: {admin_count}\n'
        f'Chat ID: {chat.id}'
    )

    async def on_chat_member(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
        result = update.my_chat_member

        old_status = result.old_chat_member.status
        new_status = result.new_chat_member.status
        chat = result.chat

        if old_status == "left" and new_status == "member":
            print(f'Bot added to: {chat.title} ({chat.id})')

        self.group_settings[chat.id] = {
            "price_alerts": False,
            "welcome_message": True,
            "language": "en"
        }

```

```
elif old_status == "member" and new_status == "left":  
    print(f"Bot removed from: {chat.title} ({chat.id})")
```

```
if chat.id in self.group_settings:  
    del self.group_settings[chat.id]
```

```
elif new_status == "administrator":  
    print(f"Bot promoted to admin in: {chat.title}")
```

```
async def on_new_member(self, update: Update, context:  
ContextTypes.DEFAULT_TYPE):  
    chat = update.effective_chat  
    new_members = update.message.new_chat_members
```

```
    settings = self.group_settings.get(chat.id, {})
```

```
    if not settings.get("welcome_message", True):  
        return
```

```
    for member in new_members:  
        if member.is_bot:  
            continue
```

```
    await update.message.reply_text(  
        f"Welcome {member.first_name} to {chat.title}!\n\n"  
        f"Use /help to see available commands."  
    )
```

```
def run(self):  
    self.application.run_polling()
```

SECTION 7: ERROR HANDLING AND RESILIENCE

Telegram bots fail in predictable ways. Network timeouts, rate limits, invalid tokens, blocked users. Robust error handling keeps your bot running when things go wrong.

Common errors and their causes:

Telegram flood limits trigger when you send too many messages too fast. The API returns error code 429 with a `retry_after` value. Respect it or face longer blocks.

Chat not found happens when users block your bot or delete their account. The `chat_id` becomes invalid. Handle gracefully without crashing.

Message too long triggers when text exceeds 4096 characters. Split long messages or truncate them.

```
from telegram import Update
from telegram.ext import Application, CommandHandler,
ContextTypes
from telegram.error import (
    TelegramError, Forbidden, NetworkError,
    BadRequest, TimedOut, RetryAfter
)
import asyncio
import logging

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

class ResilientBot:

    def __init__(self, token):
        self.application = Application.builder().token(token).build()
        self.blocked_users = set()
        self.setup_handlers()

    def setup_handlers(self):
        self.application.add_handler(
            CommandHandler("start", self.cmd_start)
        )
        self.application.add_handler(
            CommandHandler("broadcast", self.cmd_broadcast)
        )

        self.application.add_error_handler(self.error_handler)
```

```
async def cmd_start(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
    await self.safe_send_message(
        update.effective_chat.id,
        "Bot started successfully!"
    )
```

```
async def cmd_broadcast(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
    user_ids = [123, 456, 789]
    message = "Important announcement!"
```

```
    success = 0
    failed = 0
```

```
    for user_id in user_ids:
        if user_id in self.blocked_users:
            continue
```

```
    result = await self.safe_send_message(user_id, message)
```

```
    if result:
        success += 1
    else:
        failed += 1
```

```
    await asyncio.sleep(0.05)
```

```
    await update.message.reply_text(
        f"Broadcast complete\n"
        f"Success: {success}\n"
        f"Failed: {failed}"
    )
```

```
async def safe_send_message(self, chat_id, text, retry_count=3):
    for attempt in range(retry_count):
        try:
            if len(text) > 4096:
                chunks = self.split_message(text)
```

```
for chunk in chunks:
    await self.application.bot.send_message(
        chat_id = chat_id,
        text = chunk,
        parse_mode = "HTML"
    )
    await asyncio.sleep(0.1)
else:
    await self.application.bot.send_message(
        chat_id = chat_id,
        text = text,
        parse_mode = "HTML"
    )
    return True

except Forbidden as e:
    logger.warning(f"User {chat_id} blocked bot: {e}")
    self.blocked_users.add(chat_id)
    return False

except RetryAfter as e:
    logger.warning(f"Rate limited. Retry after {e.retry_after}s")
    await asyncio.sleep(e.retry_after)

except Timeout as e:
    logger.warning(f"Timeout on attempt {attempt + 1}: {e}")
    await asyncio.sleep(1)

except NetworkError as e:
    logger.error(f"Network error: {e}")
    await asyncio.sleep(2 ** attempt)

except BadRequest as e:
    logger.error(f"Bad request for {chat_id}: {e}")

if "chat not found" in str(e).lower():
    self.blocked_users.add(chat_id)

return False
```

```
except TelegramError as e:
    logger.error(f"Telegram error: {e}")
    await asyncio.sleep(1)
```

```
return False
```

```
def split_message(self, text, max_length = 4096):
    chunks = []
```

```
    while text:
        if len(text) <= max_length:
            chunks.append(text)
            break
```

```
    split_point = text.rfind("\n", 0, max_length)
```

```
    if split_point == -1:
        split_point = text.rfind(" ", 0, max_length)
```

```
    if split_point == -1:
        split_point = max_length
```

```
    chunks.append(text[:split_point])
    text = text[split_point:].lstrip()
```

```
    return chunks
```

```
    async def error_handler(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
        error = context.error
```

```
        logger.error(f"Update {update} caused error {error}")
```

```
        if isinstance(error, Forbidden):
            if update and update.effective_user:
                self.blocked_users.add(update.effective_user.id)
            return
```

```
        if isinstance(error, NetworkError):
            logger.warning("Network error - will retry automatically")
```

```
return
```

```
if isinstance(error, BadRequest):  
    logger.error(f"Bad request: {error}")  
return
```

```
if update and update.effective_message:  
    try:  
        await update.effective_message.reply_text(  
            "An error occurred. Please try again later."  
        )  
    except TelegramError:  
        pass
```

```
def run(self):  
    self.application.run_polling(  
        drop_pending_updates = True,  
        allowed_updates = Update.ALL_TYPES  
    )
```

SECTION 8: RATE LIMITING AND THROTTLING

Telegram enforces rate limits. Exceed them and your messages get delayed or dropped. Understanding and respecting these limits is essential for reliable bots.

The limits are roughly:

- 30 messages per second to different chats
- 20 messages per minute to the same group
- 1 message per second to the same user

A proper throttling system prevents limit violations:

```
import asyncio  
from collections import defaultdict  
from datetime import datetime, timedelta  
import time
```

```
class RateLimiter:
```

```

def __init__(self):
    self.global_tokens = 30
    self.global_last_update = time.time()
    self.chat_timestamps = defaultdict(list)
    self.user_timestamps = defaultdict(list)

    self.global_rate = 30
    self.chat_rate = 20
    self.chat_window = 60
    self.user_rate = 1
    self.user_window = 1

    async def acquire(self, chat_id, is_group=False):
        now = time.time()
        elapsed = now - self.global_last_update
        self.global_tokens = min(
            self.global_rate,
            self.global_tokens + elapsed * self.global_rate
        )
        self.global_last_update = now

        if self.global_tokens < 1:
            wait_time = (1 - self.global_tokens) / self.global_rate
            await asyncio.sleep(wait_time)
            self.global_tokens = 1

        self.global_tokens -= 1

        if is_group:
            cutoff = now - self.chat_window
            self.chat_timestamps[chat_id] = [
                ts for ts in self.chat_timestamps[chat_id]
                if ts > cutoff
            ]

        if len(self.chat_timestamps[chat_id]) >= self.chat_rate:
            oldest = self.chat_timestamps[chat_id][0]
            wait_time = (oldest + self.chat_window) - now
            if wait_time > 0:

```



```
await asyncio.sleep(wait_time)
```

```
self.chat_timestamps[chat_id].append(time.time())
```

```
else:
```

```
cutoff = now - self.user_window
```

```
self.user_timestamps[chat_id] = [
```

```
ts for ts in self.user_timestamps[chat_id]
```

```
if ts > cutoff
```

```
]
```

```
if len(self.user_timestamps[chat_id]) >= self.user_rate:
```

```
oldest = self.user_timestamps[chat_id][0]
```

```
wait_time = (oldest + self.user_window) - now
```

```
if wait_time > 0:
```

```
await asyncio.sleep(wait_time)
```

```
self.user_timestamps[chat_id].append(time.time())
```

```
class MessageQueue:
```

```
def __init__(self, bot, rate_limiter):
```

```
self.bot = bot
```

```
self.rate_limiter = rate_limiter
```

```
self.queue = asyncio.Queue()
```

```
self.running = False
```

```
async def start(self):
```

```
self.running = True
```

```
asyncio.create_task(self.process_queue())
```

```
async def stop(self):
```

```
self.running = False
```

```
async def enqueue(self, chat_id, text, is_group=False, priority=0):
```

```
await self.queue.put({
```

```
"chat_id": chat_id,
```

```
"text": text,
```

```
"is_group": is_group,
```

```
"priority": priority,
```

```
"timestamp": time.time()
})
```

```
async def process_queue(self):
while self.running:
try:
message = await asyncio.wait_for(
self.queue.get(),
timeout=1.0
)
```

```
await self.rate_limiter.acquire(
message["chat_id"],
message["is_group"]
)
```

```
try:
await self.bot.send_message(
chat_id=message["chat_id"],
text=message["text"]
)
except Exception as e:
print(f"Send failed: {e}")
```

```
self.queue.task_done()
```

```
except asyncio.TimeoutError:
continue
```

```
class ThrottledBot:
```

```
def __init__(self, token):
from telegram.ext import Application
self.application = Application.builder().token(token).build()
self.rate_limiter = RateLimiter()
self.message_queue = None
```

```
async def post_init(self, application):
self.message_queue = MessageQueue(
```

```

application.bot,
self.rate_limiter
)
await self.message_queue.start()

async def send_throttled(self, chat_id, text, is_group=False):
await self.message_queue.enqueue(chat_id, text, is_group)

async def broadcast_to_users(self, user_ids, message):
for user_id in user_ids:
await self.send_throttled(user_id, message, is_group=False)

async def broadcast_to_groups(self, group_ids, message):
for group_id in group_ids:
await self.send_throttled(group_id, message, is_group=True)

```

SECTION 9: WEBHOOK DEPLOYMENT WITH FASTAPI

Production bots typically use webhooks. FastAPI provides an excellent foundation - async support, automatic documentation, and high performance. This setup handles real traffic.

```

from fastapi import FastAPI, Request, HTTPException
from telegram import Update, Bot
from telegram.ext import Application, CommandHandler
import uvicorn
import asyncio
import os

```

```

app = FastAPI(title="Crypto Bot Webhook")

```

```

class WebhookApplication:

```

```

    def __init__(self):
        self.token = os.getenv("BOT_TOKEN")
        self.webhook_url = os.getenv("WEBHOOK_URL")
        self.bot = Bot(self.token)
        self.application = None

```

```
async def setup(self):
    self.application = (
        Application.builder()
        .token(self.token)
        .build()
    )
```

```
self.application.add_handler(
    CommandHandler("start", self.handle_start)
)
self.application.add_handler(
    CommandHandler("price", self.handle_price)
)
```

```
await self.application.initialize()
```

```
await self.bot.set_webhook(
    url=f'{self.webhook_url}/webhook',
    allowed_updates=["message", "callback_query"],
    drop_pending_updates=True,
    max_connections=100
)
```

```
async def shutdown(self):
    await self.bot.delete_webhook()
    await self.application.shutdown()
```

```
async def process_update(self, update_data: dict):
    update = Update.de_json(update_data, self.bot)
    await self.application.process_update(update)
```

```
async def handle_start(self, update, context):
    await update.message.reply_text(
        "Welcome to Crypto Bot!\n"
        "Running on webhook mode."
    )
```

```
async def handle_price(self, update, context):
    await update.message.reply_text(
        "ETH: $2,345.67\n"
```

```
"BTC: $43,210.00"
```

```
)
```

```
webhook_app = WebhookApplication()
```

```
@app.on_event("startup")
```

```
async def startup():
```

```
    await webhook_app.setup()
```

```
@app.on_event("shutdown")
```

```
async def shutdown():
```

```
    await webhook_app.shutdown()
```

```
@app.post("/webhook")
```

```
async def webhook(request: Request):
```

```
    try:
```

```
        update_data = await request.json()
```

```
        await webhook_app.process_update(update_data)
```

```
        return {"status": "ok"}
```

```
    except Exception as e:
```

```
        print(f"Webhook error: {e}")
```

```
        raise HTTPException(status_code=500, detail=str(e))
```

```
@app.get("/health")
```

```
async def health():
```

```
    return {
```

```
        "status": "healthy",
```

```
        "bot_username": webhook_app.bot.username if webhook_app.bot  
    else None
```

```
    }
```

```
@app.get("/webhook/info")
```

```
async def webhook_info():
```

```
    info = await webhook_app.bot.get_webhook_info()
```

```
    return {
```

```
        "url": info.url,
```

```
        "pending_update_count": info.pending_update_count,
```

```
        "last_error_date": info.last_error_date,
```

```
        "last_error_message": info.last_error_message,
```

```
        "max_connections": info.max_connections
```

```
}
```

The deployment uses environment variables:

```
BOT_TOKEN=your_bot_token  
WEBHOOK_URL=https://your-domain.com  
PORT=8000
```

Running with uvicorn:

```
uvicorn main:app --host 0.0.0.0 --port 8000
```

SECTION 10: TESTING BOT HANDLERS

Testing bots is tricky because they interact with external services. Mock the Telegram API to test your logic without making real requests.

```
import pytest  
from unittest.mock import AsyncMock, MagicMock, patch  
from telegram import Update, User, Chat, Message
```

```
class TestBotHandlers:
```

```
    def create_update(self, text, user_id=123, chat_id=123,  
is_command=False):  
        user = User(  
            id=user_id,  
            first_name="Test",  
            is_bot=False,  
            username="testuser"  
        )
```

```
        chat = Chat(  
            id=chat_id,  
            type="private"  
        )
```

```
        message = MagicMock(spec=Message)
```

```
message.text = text
message.from_user = user
message.chat = chat
message.reply_text = AsyncMock()
message.edit_text = AsyncMock()
```

```
if is_command:
    message.entities = [MagicMock(type="bot_command", offset=0,
length=len(text.split()[0]))]
```

```
update = MagicMock(spec=Update)
update.message = message
update.effective_user = user
update.effective_chat = chat
update.effective_message = message
update.callback_query = None
```

```
return update
```

```
def create_context(self, args=None, user_data=None):
    context = MagicMock()
    context.args = args or []
    context.user_data = user_data or {}
    context.bot = AsyncMock()
    context.bot.send_message = AsyncMock()
    return context
```

```
@pytest.mark.asyncio
async def test_start_command(self):
    from your_bot import MessageBot
```

```
bot = MessageBot("fake_token")
update = self.create_update("/start", is_command=True)
context = self.create_context()
```

```
await bot.cmd_start(update, context)
```

```
update.message.reply_text.assert_called_once()
call_args = update.message.reply_text.call_args
assert "Welcome" in call_args[0][0]
```

```
@pytest.mark.asyncio
async def test_balance_unregistered_user(self):
    from your_bot import MessageBot
```

```
    bot = MessageBot("fake_token")
    update = self.create_update("/balance", user_id = 999,
is_command = True)
    context = self.create_context()
```

```
    await bot.cmd_balance(update, context)
```

```
    update.message.reply_text.assert_called_once()
    call_args = update.message.reply_text.call_args
    assert "start first" in call_args[0][0].lower()
```

```
@pytest.mark.asyncio
async def test_send_invalid_amount(self):
    from your_bot import MessageBot
```

```
    bot = MessageBot("fake_token")
    update = self.create_update("/send abc 0x123",
is_command = True)
    context = self.create_context(args=["abc", "0x123"])
```

```
    await bot.cmd_send(update, context)
```

```
    update.message.reply_text.assert_called_once()
    call_args = update.message.reply_text.call_args
    assert "invalid" in call_args[0][0].lower()
```

```
@pytest.mark.asyncio
async def test_send_invalid_address(self):
    from your_bot import MessageBot
```

```
    bot = MessageBot("fake_token")
    bot.user_data[123] = {"balance": 10.0}
```

```
    update = self.create_update("/send 1.0 invalid",
is_command = True)
```



```
context = self.create_context(args=["1.0", "invalid"])
```

```
await bot.cmd_send(update, context)
```

```
call_args = update.message.reply_text.call_args
assert "invalid" in call_args[0][0].lower()
```

```
class TestRateLimiter:
```

```
@pytest.mark.asyncio
async def test_global_rate_limit(self):
    from your_bot import RateLimiter
    import time
```

```
    limiter = RateLimiter()
    limiter.global_tokens = 0
    limiter.global_last_update = time.time()
```

```
    start = time.time()
    await limiter.acquire(123, is_group=False)
    elapsed = time.time() - start
```

```
    assert elapsed > 0.01
```

```
@pytest.mark.asyncio
async def test_chat_rate_limit(self):
    from your_bot import RateLimiter
    import time
```

```
    limiter = RateLimiter()
    limiter.chat_rate = 2
    limiter.chat_window = 1
```

```
    await limiter.acquire(123, is_group=True)
    await limiter.acquire(123, is_group=True)
```

```
    start = time.time()
    await limiter.acquire(123, is_group=True)
    elapsed = time.time() - start
```

```
assert elapsed > 0.5
```

```
class TestMessageSplitting:
```

```
    def test_split_long_message(self):  
        from your_bot import ResilientBot
```

```
        bot = ResilientBot("fake_token")
```

```
        long_text = "A" * 5000  
        chunks = bot.split_message(long_text)
```

```
        assert len(chunks) == 2  
        assert all(len(chunk) <= 4096 for chunk in chunks)  
        assert "".join(chunks) == long_text
```

```
    def test_split_at_newline(self):  
        from your_bot import ResilientBot
```

```
        bot = ResilientBot("fake_token")
```

```
        text = "A" * 4000 + "\n" + "B" * 100  
        chunks = bot.split_message(text)
```

```
        assert len(chunks) == 2  
        assert chunks[0] == "A" * 4000  
        assert chunks[1] == "B" * 100
```

```
    def test_no_split_needed(self):  
        from your_bot import ResilientBot
```

```
        bot = ResilientBot("fake_token")
```

```
        short_text = "Hello world"  
        chunks = bot.split_message(short_text)
```

```
        assert len(chunks) == 1  
        assert chunks[0] == short_text
```

Running tests:

```
pytest test_bot.py -v --asyncio-mode=auto
```

CHAPTER SUMMARY

Telegram bot fundamentals provide the foundation for everything that follows. You now understand:

The Bot API is HTTP-based. Every library wraps the same endpoints. Understanding the raw API helps debug when abstractions fail.

Polling works anywhere, webhooks work better at scale. Start with polling for development, deploy webhooks for production.

Message handlers route updates to your code. Commands, text, photos - each type gets its handler. Error handlers catch what falls through.

Inline keyboards create rich interfaces. Callback queries let buttons trigger actions. The chat interface becomes an application.

Conversation handlers manage multi-step flows. State machines track where users are in a process. Timeouts handle abandoned conversations.

Groups behave differently than private chats. Privacy mode, admin permissions, and addressing affect how bots operate.

Error handling is essential. Rate limits, blocked users, network failures - all must be handled gracefully.

Rate limiting prevents API abuse. Token buckets and message queues ensure you stay within Telegram's limits.

Testing requires mocking. Create fake updates and contexts to test

handler logic without real API calls.

Next chapter: Wallet Integration. We connect Telegram users to blockchain wallets.

CHAPTER 1: TELEGRAM ECOSYSTEM 2025

Understanding the complete Telegram landscape before writing code. This chapter maps every component - Bot API, Mini Apps, TON blockchain, Telegram Stars, trading infrastructure. You'll understand how pieces connect, what's mandatory, and where opportunities exist.

SECTION 1: THE TELEGRAM PLATFORM ARCHITECTURE

Telegram is not just a messaging app. It's a platform with multiple interconnected systems. Understanding this architecture prevents building the wrong thing in the wrong place.

The Core Messaging Layer handles text, media, and communication. Users send messages. Groups aggregate communities. Channels broadcast to audiences. This existed from the beginning and remains the foundation.

The Bot Layer adds automation on top of messaging. Bots are special accounts controlled by code. They receive messages, send responses, and interact through keyboards and callbacks. The Bot API exposes this functionality via HTTP endpoints.

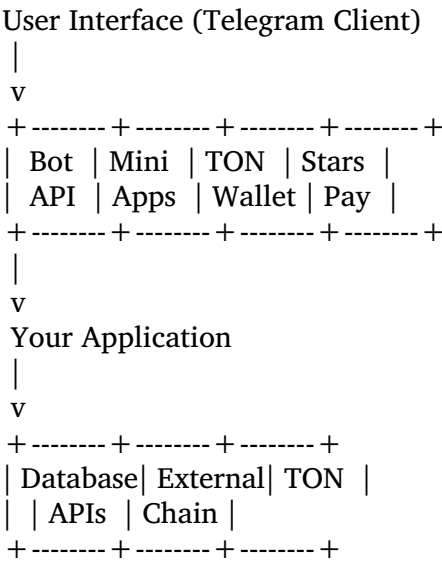
The Mini Apps Layer brings web applications into Telegram. These are full web apps rendered within the Telegram interface. Users never leave the app. Mini Apps access device features, handle payments, and integrate with wallets.

The TON Layer provides blockchain infrastructure. Wallets, tokens, smart contracts, decentralized applications. Since January 2025, this is the exclusive blockchain layer for Telegram.

The Payments Layer connects everything financially. Telegram Stars for digital goods. Toncoin for crypto transactions. Both integrate

with bots and Mini Apps through dedicated APIs.

This diagram shows the relationships:



Your application sits at the intersection. It uses Bot API for chat interactions, Mini Apps for rich interfaces, TON Wallet for crypto operations, and Stars for monetization. Understanding each layer's capabilities determines what you can build.

SECTION 2: BOT API 8.X DEEP DIVE

The Bot API in 2025 handles far more than simple message responses. Let's examine the major feature categories.

Message Types and Media:

Bots can send and receive text, photos, videos, audio, documents, animations, stickers, voice messages, video notes, locations, contacts, polls, and dice. Each type has specific methods and parameters.

The message entity system parses text for mentions, hashtags, URLs,

bot commands, code blocks, and formatting. Your bot receives parsed entities alongside raw text.

```
import asyncio
from telegram import Update, Bot
from telegram.ext import Application, MessageHandler, filters
```

```
class MessageAnalyzer:
```

```
    def __init__(self, token):
        self.app = Application.builder().token(token).build()
        self.app.add_handler(
            MessageHandler(filters.ALL, self.analyze_message)
        )
```

```
    async def analyze_message(self, update: Update, context):
        message = update.effective_message
```

```
        analysis = {
            "message_id": message.message_id,
            "date": message.date.isoformat(),
            "chat_type": update.effective_chat.type,
            "has_text": bool(message.text),
            "has_photo": bool(message.photo),
            "has_video": bool(message.video),
            "has_document": bool(message.document),
            "has_audio": bool(message.audio),
            "has_voice": bool(message.voice),
            "has_sticker": bool(message.sticker),
            "has_location": bool(message.location),
            "has_contact": bool(message.contact),
            "has_poll": bool(message.poll),
            "has_dice": bool(message.dice),
            "has_animation": bool(message.animation),
            "has_forward": bool(message.forward_date),
            "has_reply": bool(message.reply_to_message),
            "entities": []
        }
```

```
        if message.entities:
```

```

for entity in message.entities:
    analysis["entities"].append({
        "type": entity.type,
        "offset": entity.offset,
        "length": entity.length,
        "url": entity.url if entity.type == "text_link" else None,
        "user": entity.user.id if entity.user else None
    })

```

```

print(f'Message analysis: {analysis}')

```

```

def run(self):
    self.app.run_polling()

```

Message Streaming (Bot API 8.0 +):

The `sendMessageDraft` method enables streaming responses. Instead of waiting for complete generation, bots can send partial content that updates in real-time. This transforms user experience for AI-powered bots.

```

class StreamingBot:

```

```

    def __init__(self, token):
        self.bot = Bot(token)

```

```

    async def stream_response(self, chat_id, generator_func):
        draft_message = None
        accumulated_text = ""

```

```

        async for chunk in generator_func():
            accumulated_text += chunk

```

```

        if draft_message is None:
            draft_message = await self.bot.send_message_draft(
                chat_id=chat_id,
                text=accumulated_text
            )
        else:

```

```
await self.bot.edit_message_draft(
chat_id = chat_id,
message_id = draft_message.message_id,
text = accumulated_text
)
```

```
await asyncio.sleep(0.1)
```

```
await self.bot.finalize_message_draft(
chat_id = chat_id,
message_id = draft_message.message_id
)
```

```
return draft_message
```

```
async def ai_response_generator(self, prompt):
response_parts = [
"Let me ",
"analyze ",
"your ",
"request...\n\n",
"Based on ",
"current ",
"market ",
"conditions:\n",
"- ETH shows ",
"bullish ",
"momentum\n",
"- Volume ",
"increasing ",
"steadily"
]
```

```
for part in response_parts:
yield part
await asyncio.sleep(0.2)
```

Business 2.0 Features:

Bots can now manage Telegram Business accounts entirely. This means automated customer service, CRM integration, and business workflow automation.

```
class BusinessBot:
```

```
    def __init__(self, token):  
        self.bot = Bot(token)
```

```
    async def manage_business_account(self, business_connection_id):  
        await self.bot.set_business_account_name(  
            business_connection_id=business_connection_id,  
            name="Crypto Trading Support"  
        )
```

```
        await self.bot.set_business_account_bio(  
            business_connection_id=business_connection_id,  
            bio="24/7 automated trading support. DM for help."  
        )
```

```
    async def handle_business_message(self, update: Update):  
        business_message = update.business_message
```

```
        sender = business_message.from_user  
        chat = business_message.chat  
        text = business_message.text
```

```
        if "price" in text.lower():  
            await self.bot.send_message(  
                chat_id=chat.id,  
                text="Current prices:\nBTC: $43,210\nETH: $2,345\nTON:  
$5.67",  
                business_connection_id=update.business_connection_id  
            )
```

```
        await self.bot.mark_business_message_as_read(  
            business_connection_id=update.business_connection_id,  
            message_id=business_message.message_id  
        )
```

```

async def delete_business_messages(self, connection_id,
message_ids):
    await self.bot.delete_business_messages(
        business_connection_id=connection_id,
        message_ids=message_ids
    )

```

Reactions and Interactive Features:

Bots can now add reactions to messages, including service messages. This enables acknowledgment, voting, and engagement features.

```

class ReactiveBot:

```

```

    def __init__(self, token):
        self.bot = Bot(token)

```

```

    async def react_to_message(self, chat_id, message_id, emoji):
        await self.bot.set_message_reaction(
            chat_id=chat_id,
            message_id=message_id,
            reaction=[{"type": "emoji", "emoji": emoji}]
        )

```

```

    async def handle_trade_confirmation(self, update: Update,
trade_success):
        message = update.effective_message

```

```

        if trade_success:
            await self.react_to_message(
                update.effective_chat.id,
                message.message_id,
                "check"
            )
        else:
            await self.react_to_message(
                update.effective_chat.id,
                message.message_id,

```

```
"cross"  
)
```

SECTION 3: MINI APPS 2.0 ARCHITECTURE

Mini Apps evolved from simple web views to full applications. Understanding the architecture helps you design appropriately.

The Web App Container:

Mini Apps run in a WebView within Telegram. They're standard web applications (HTML, CSS, JavaScript) with access to Telegram-specific APIs through the `window.Telegram.WebApp` object.

The initialization flow:

1. User taps bot button or link
2. Telegram opens WebView
3. Your web app loads
4. App calls `Telegram.WebApp.ready()`
5. App receives initialization data
6. App can access Telegram features

This JavaScript shows proper initialization:

```
class TelegramMiniApp {  
  
  constructor() {  
    this.tg = window.Telegram.WebApp;  
    this.user = null;  
    this.initData = null;  
    this.init();  
  }  
  
  init() {  
    this.tg.ready();  
  
    this.tg.expand();  
  }  
}
```

```
this.initData = this.tg.initDataUnsafe;  
this.user = this.initData.user;
```

```
this.applyTheme();
```

```
this.tg.onEvent('themeChanged', () => this.applyTheme());  
this.tg.onEvent('viewportChanged', (e) =>  
this.handleViewport(e));  
this.tg.onEvent('mainButtonClicked', () =>  
this.handleMainButton());
```

```
console.log('Mini App initialized');  
console.log('User:', this.user);  
console.log('Platform:', this.tg.platform);  
console.log('Version:', this.tg.version);  
}
```

```
applyTheme() {  
  const theme = this.tg.themeParams;
```

```
  document.documentElement.style.setProperty(  
    '--tg-bg-color',  
    theme.bg_color || '#ffffff'  
  );  
  document.documentElement.style.setProperty(  
    '--tg-text-color',  
    theme.text_color || '#000000'  
  );  
  document.documentElement.style.setProperty(  
    '--tg-hint-color',  
    theme.hint_color || '#999999'  
  );  
  document.documentElement.style.setProperty(  
    '--tg-link-color',  
    theme.link_color || '#2481cc'  
  );  
  document.documentElement.style.setProperty(  
    '--tg-button-color',  
    theme.button_color || '#2481cc'  
  );
```

```
document.documentElement.style.setProperty(
  '--tg-button-text-color',
  theme.button_text_color || '#ffffff'
);
}
```

```
handleViewport(event) {
  console.log('Viewport changed:', event);
  console.log('Is expanded:', event.isStateStable);
}
```

```
handleMainButton() {
  console.log('Main button clicked');
}
```

```
showMainButton(text, callback) {
  this.tg.MainButton.setText(text);
  this.tg.MainButton.show();
  this.mainButtonCallback = callback;
}
```

```
hideMainButton() {
  this.tg.MainButton.hide();
}
```

```
sendData(data) {
  this.tg.sendData(JSON.stringify(data));
}
```

```
close() {
  this.tg.close();
}
}
```

```
const app = new TelegramMiniApp();
```

Full-Screen Mode:

Mini Apps 2.0 support full-screen experiences. This enables games,

video players, and immersive applications.

```
class FullScreenMiniApp extends TelegramMiniApp {
```

```
  constructor() {  
    super();  
    this.isFullScreen = false;  
  }
```

```
  async requestFullScreen() {  
    if (!this.tg.isVersionAtLeast('8.0')) {  
      console.log('Full screen not supported');  
      return false;  
    }
```

```
    try {  
      await this.tg.requestFullscreen();  
      this.isFullScreen = true;  
      this.handleFullScreenEnter();  
      return true;  
    } catch (error) {  
      console.error('Failed to enter fullscreen:', error);  
      return false;  
    }  
  }
```

```
  async exitFullScreen() {  
    try {  
      await this.tg.exitFullscreen();  
      this.isFullScreen = false;  
      this.handleFullScreenExit();  
    } catch (error) {  
      console.error('Failed to exit fullscreen:', error);  
    }  
  }
```

```
  handleFullScreenEnter() {  
    document.body.classList.add('fullscreen-mode');
```

```
    const safeArea = this.tg.safeAreaInset;
```

```
const contentSafeArea = this.tg.contentSafeAreaInset;
```

```
document.documentElement.style.setProperty(
  '--safe-area-top',
  safeArea.top + 'px'
);
document.documentElement.style.setProperty(
  '--safe-area-bottom',
  safeArea.bottom + 'px'
);
document.documentElement.style.setProperty(
  '--safe-area-left',
  safeArea.left + 'px'
);
document.documentElement.style.setProperty(
  '--safe-area-right',
  safeArea.right + 'px'
);
}
```

```
handleFullScreenExit() {
  document.body.classList.remove('fullscreen-mode');
}
```

```
lockOrientation(orientation) {
  if (this.tg.isVersionAtLeast('8.0')) {
    this.tg.lockOrientation(orientation);
  }
}
```

```
unlockOrientation() {
  if (this.tg.isVersionAtLeast('8.0')) {
    this.tg.unlockOrientation();
  }
}
```

Device Sensors:

Mini Apps can access accelerometer, gyroscope, and device orientation. This enables motion-controlled games and fitness applications.

```
class SensorMiniApp extends TelegramMiniApp {

  constructor() {
    super();
    this.accelerometer = null;
    this.gyroscope = null;
    this.deviceOrientation = null;
  }

  startAccelerometer(refreshRate = 100) {
    if (!this.tg.Accelerometer) {
      console.log('Accelerometer not available');
      return;
    }

    this.accelerometer = this.tg.Accelerometer;
    this.accelerometer.start({ refresh_rate: refreshRate });

    this.accelerometer.onData((data) => {
      this.handleAccelerometerData(data);
    });
  }

  stopAccelerometer() {
    if (this.accelerometer) {
      this.accelerometer.stop();
    }
  }

  handleAccelerometerData(data) {
    const x = data.x;
    const y = data.y;
    const z = data.z;

    console.log(` Accelerometer: x=${x}, y=${y}, z=${z}`);
  }
}
```



```
this.onAccelerometerUpdate(x, y, z);  
}
```

```
onAccelerometerUpdate(x, y, z) {  
}
```

```
startGyroscope(refreshRate = 100) {  
  if (!this.tg.Gyroscope) {  
    console.log('Gyroscope not available');  
    return;  
  }  
}
```

```
this.gyroscope = this.tg.Gyroscope;  
this.gyroscope.start({ refresh_rate: refreshRate });
```

```
this.gyroscope.onData((data) => {  
  this.handleGyroscopeData(data);  
});  
}
```

```
handleGyroscopeData(data) {  
  const x = data.x;  
  const y = data.y;  
  const z = data.z;
```

```
this.onGyroscopeUpdate(x, y, z);  
}
```

```
onGyroscopeUpdate(x, y, z) {  
}
```

```
startDeviceOrientation(refreshRate = 100, absolute = false) {  
  if (!this.tg.DeviceOrientation) {  
    console.log('Device orientation not available');  
    return;  
  }  
}
```

```
this.deviceOrientation = this.tg.DeviceOrientation;  
this.deviceOrientation.start({  
  refresh_rate: refreshRate,
```

```
need_absolute: absolute
});
```

```
this.deviceOrientation.onData((data) => {
  this.handleOrientationData(data);
});
}
```

```
handleOrientationData(data) {
  const alpha = data.alpha;
  const beta = data.beta;
  const gamma = data.gamma;
```

```
this.onOrientationUpdate(alpha, beta, gamma);
}
```

```
onOrientationUpdate(alpha, beta, gamma) {
}
}
```

Storage APIs:

Mini Apps have access to both DeviceStorage (persistent) and SecureStorage (encrypted). This enables offline functionality and secure credential storage.

```
class StorageMiniApp extends TelegramMiniApp {
```

```
  constructor() {
    super();
    this.deviceStorage = this.tg.DeviceStorage;
    this.secureStorage = this.tg.SecureStorage;
  }
```

```
  async saveToDeviceStorage(key, value) {
    try {
      await this.deviceStorage.setItem(key, JSON.stringify(value));
      return true;
    } catch (error) {
```

```
console.error('DeviceStorage save failed:', error);  
return false;  
}  
}
```

```
async getFromDeviceStorage(key) {  
  try {  
    const value = await this.deviceStorage.getItem(key);  
    return value ? JSON.parse(value) : null;  
  } catch (error) {  
    console.error('DeviceStorage get failed:', error);  
    return null;  
  }  
}
```

```
async removeFromDeviceStorage(key) {  
  try {  
    await this.deviceStorage.removeItem(key);  
    return true;  
  } catch (error) {  
    console.error('DeviceStorage remove failed:', error);  
    return false;  
  }  
}
```

```
async saveToSecureStorage(key, value) {  
  try {  
    await this.secureStorage.setItem(key, value);  
    return true;  
  } catch (error) {  
    console.error('SecureStorage save failed:', error);  
    return false;  
  }  
}
```

```
async getFromSecureStorage(key) {  
  try {  
    return await this.secureStorage.getItem(key);  
  } catch (error) {  
    console.error('SecureStorage get failed:', error);  
  }  
}
```

```

return null;
}
}

async saveUserPreferences(prefs) {
await this.saveToDeviceStorage('user_preferences', prefs);
}

async getUserPreferences() {
return await this.getFromDeviceStorage('user_preferences') || {
theme: 'auto',
currency: 'USD',
notifications: true
};
}

async saveApiToken(token) {
await this.saveToSecureStorage('api_token', token);
}

async getApiToken() {
return await this.getFromSecureStorage('api_token');
}
}

```

SECTION 4: TELEGRAM STARS ECONOMY

Telegram Stars (currency tag XTR) are the mandatory payment method for digital goods in Mini Apps. Understanding this economy is essential for monetization.

Why Stars are Mandatory:

Apple and Google app store policies require in-app purchases to use their payment systems for digital goods. Telegram negotiated a solution: Telegram Stars. All digital goods sold through Mini Apps displayed on mobile must use Stars. No exceptions.

Users purchase Stars through standard in-app purchases (Apple Pay,

Google Pay) or via @PremiumBot. Then they spend Stars in bots and Mini Apps. Developers earn Stars and can withdraw them to Toncoin.

The Star Lifecycle:

1. User buys Stars (fiat to Stars via app stores)
2. User spends Stars in your bot/Mini App
3. You receive Stars in your balance
4. Stars have 21-day hold period
5. You withdraw Stars to TON via Fragment
6. You sell TON for fiat if desired

Creating Star Invoices:

```
class StarPaymentBot:
```

```
    def __init__(self, token):
        self.bot = Bot(token)
```

```
    async def create_star_invoice(
        self,
        chat_id,
        title,
        description,
        payload,
        star_amount
    ):
        await self.bot.send_invoice(
            chat_id=chat_id,
            title=title,
            description=description,
            payload=payload,
            currency="XTR",
            prices=[{
                "label": title,
                "amount": star_amount
            }]
        )
```

```
async def create_invoice_link(
    self,
    title,
    description,
    payload,
    star_amount
):
    link = await self.bot.create_invoice_link(
        title=title,
        description=description,
        payload=payload,
        currency="XTR",
        prices=[{
            "label": title,
            "amount": star_amount
        }]
    )
    return link
```

```
async def handle_pre_checkout(self, update: Update, context):
    query = update.pre_checkout_query
```

```
    payload = query.invoice_payload
    user_id = query.from_user.id
    total_amount = query.total_amount
```

```
    is_valid = await self.validate_purchase(payload, user_id)
```

```
    if is_valid:
        await query.answer(ok=True)
    else:
        await query.answer(
            ok=False,
            error_message="This item is no longer available"
        )
```

```
async def validate_purchase(self, payload, user_id):
    return True
```

```
async def handle_successful_payment(self, update: Update, context):
```

```
payment = update.message.successful_payment
```

```
payload = payment.invoice_payload  
total_amount = payment.total_amount  
telegram_payment_charge_id =  
payment.telegram_payment_charge_id  
user_id = update.effective_user.id
```

```
await self.fulfill_order(  
    user_id=user_id,  
    payload=payload,  
    amount=total_amount,  
    charge_id=telegram_payment_charge_id  
)
```

```
await update.message.reply_text(  
    f'Payment successful! You paid {total_amount} Stars.\n'  
    f'Your purchase has been activated.'  
)
```

```
async def fulfill_order(self, user_id, payload, amount, charge_id):  
    print(f'Fulfilling order: {payload} for user {user_id}')  
    print(f'Amount: {amount} Stars, Charge ID: {charge_id}')
```

Star Subscriptions:

Recurring payments use Stars with automatic renewal. This enables premium tiers, ongoing services, and membership models.

class SubscriptionBot:

```
    def __init__(self, token):  
        self.bot = Bot(token)  
        self.subscription_tiers = {  
            "basic": {  
                "stars": 100,  
                "period": 2592000,  
                "features": ["price_alerts", "portfolio_view"]  
            },  
            "premium": {  
                "stars": 1000,  
                "period": 2592000,  
                "features": ["price_alerts", "portfolio_view", "premium_support"]  
            }  
        }
```

```

"pro": {
"stars": 500,
"period": 2592000,
"features": ["price_alerts", "portfolio_view", "trading", "analytics"]
},
"whale": {
"stars": 2000,
"period": 2592000,
"features": ["all", "priority_support", "api_access"]
}
}

```

```

async def create_subscription_link(self, tier):
tier_config = self.subscription_tiers.get(tier)
if not tier_config:
return None

```

```

link = await self.bot.create_invoice_link(
title=f'{tier.title()} Subscription',
description=f'Monthly {tier} access with: {',
'.join(tier_config['features'])}',
payload=f'subscription:{tier}',
currency="XTR",
prices=[{
"label": f'{tier.title()} Monthly',
"amount": tier_config["stars"]
}],
subscription_period = tier_config["period"]
)

```

```

return link

```

```

async def handle_subscription_payment(self, update: Update,
context):
payment = update.message.successful_payment
payload = payment.invoice_payload

```

```

if payload.startswith("subscription:"):
tier = payload.split(":")[1]
user_id = update.effective_user.id

```



```
await self.activate_subscription(user_id, tier)
```

```
await update.message.reply_text(  
f"Welcome to {tier.title()}!\n\n"  
f"Your subscription is now active and will renew automatically.\n"  
f"Manage your subscription in Settings."  
)
```

```
async def activate_subscription(self, user_id, tier):  
    print(f"Activating {tier} subscription for user {user_id}")
```

Star Withdrawal:

Earned Stars convert to Toncoin through Fragment (fragment.com).
The process:

1. Accumulate 1000+ Stars
2. Wait 21 days after earning
3. Connect wallet to Fragment
4. Initiate withdrawal
5. Receive TON in wallet
6. Sell TON on exchange if desired

The withdrawal fee is approximately 5%. Plan your pricing accordingly.

SECTION 5: TON BLOCKCHAIN INTEGRATION

TON (The Open Network) is now the exclusive blockchain for Telegram. Understanding TON architecture is required for any blockchain feature.

TON Architecture Overview:

TON differs from Ethereum significantly. Key differences:

Workchains and Shardchains: TON can theoretically handle millions

of transactions per second through sharding. The masterchain coordinates, while workchains handle execution.

Account Model: Every entity in TON is a smart contract, including user wallets. There are no "externally owned accounts" like Ethereum.

Message-Based: Contracts communicate through messages, not direct calls. This enables true asynchronous execution.

FunC Language: Smart contracts use FunC, a C-like language, compiled to TVM (TON Virtual Machine) bytecode.

TON Connect Protocol:

TON Connect is the mandatory wallet connection protocol. It enables secure communication between your app and user wallets.

```
const TonConnect = require('@tonconnect/sdk');
```

```
class TONConnector {
```

```
  constructor(manifestUrl) {  
    this.connector = new TonConnect.TonConnect({  
      manifestUrl: manifestUrl  
    });
```

```
    this.connector.onStatusChange(wallet => {  
      this.handleWalletChange(wallet);  
    });  
  }
```

```
  handleWalletChange(wallet) {  
    if (wallet) {  
      console.log('Wallet connected:', wallet.account.address);  
      this.onWalletConnected(wallet);  
    } else {  
      console.log('Wallet disconnected');  
      this.onWalletDisconnected();  
    }  
  }
```

```
}
```

```
onWalletConnected(wallet) {  
}
```

```
onWalletDisconnected() {  
}
```

```
async getWallets() {  
  const wallets = await this.connector.getWallets();  
  return wallets;  
}
```

```
async connect(walletName) {  
  const wallets = await this.getWallets();  
  const wallet = wallets.find(w => w.name === walletName);
```

```
  if (!wallet) {  
    throw new Error(`Wallet ${walletName} not found`);  
  }
```

```
  const universalLink = this.connector.connect({  
    jsBridgeKey: wallet.jsBridgeKey  
  });
```

```
  return universalLink;  
}
```

```
async connectTonkeeper() {  
  return await this.connect("Tonkeeper");  
}
```

```
isConnected() {  
  return this.connector.connected;  
}
```

```
getAccount() {  
  if (!this.connector.wallet) {  
    return null;  
  }
```

```

return {
  address: this.connector.wallet.account.address,
  chain: this.connector.wallet.account.chain,
  publicKey: this.connector.wallet.account.publicKey
};
}

```

```

async disconnect() {
  await this.connector.disconnect();
}

```

```

async sendTransaction(transaction) {
  if (!this.isConnected()) {
    throw new Error('Wallet not connected');
  }
}

```

```

const result = await this.connector.sendTransaction(transaction);
return result;
}

```

```

async sendTon(recipientAddress, amountNano, message = "") {
  const transaction = {
    validUntil: Math.floor(Date.now() / 1000) + 600,
    messages: [
      {
        address: recipientAddress,
        amount: amountNano.toString(),
        payload: message
      }
    ]
  };
}

```

```

return await this.sendTransaction(transaction);
}
}

```

TON in Mini Apps:

Combining TON Connect with Mini Apps creates powerful crypto applications:

```
class TONMiniApp extends TelegramMiniApp {

  constructor() {
    super();
    this.tonConnector = null;
  }

  async initTON(manifestUrl) {
    this.tonConnector = new TONConnector(manifestUrl);

    const savedConnection = await
    this.getFromDeviceStorage('ton_connection');
    if (savedConnection) {
      await this.tonConnector.restoreConnection();
    }
  }

  async connectWallet() {
    const wallets = await this.tonConnector.getWallets();

    const walletOptions = wallets.map(w => ({
      name: w.name,
      imageUrl: w.imageUrl,
      aboutUrl: w.aboutUrl
    }));

    this.showWalletSelector(walletOptions);
  }

  showWalletSelector(wallets) {
    console.log('Available wallets:', wallets);
  }

  async handleWalletSelection(walletName) {
    try {
      const universalLink = await
      this.tonConnector.connect(walletName);
    }
  }
}
```

```
if (this.tg.platform === 'ios' || this.tg.platform === 'android') {  
  window.location.href = universalLink;  
} else {  
  this.showQRCode(universalLink);  
}  
} catch (error) {  
  console.error('Connection failed:', error);  
}  
}
```

```
showQRCode(link) {  
  console.log('Show QR for:', link);  
}
```

```
async sendPayment(recipient, amountTON) {  
  if (!this.tonConnector.isConnected()) {  
    await this.connectWallet();  
    return;  
  }
```

```
  const amountNano = Math.floor(amountTON * 1e9);
```

```
  try {  
    const result = await this.tonConnector.sendTon(  
      recipient,  
      amountNano,  
      'Payment from Mini App'  
    );
```

```
    this.tg.showAlert('Payment sent successfully!');  
    return result;  
  } catch (error) {  
    this.tg.showAlert('Payment failed: ' + error.message);  
    throw error;  
  }  
}
```

```
getConnectedAddress() {  
  const account = this.tonConnector.getAccount();
```

```
return account ? account.address : null;
}
}
```

SECTION 6: TRADING BOT ECOSYSTEM ANALYSIS

The trading bot market provides lessons for any Telegram crypto project. Let's analyze what makes them successful.

Trojan (Unibot) Architecture:

2 million users and 24+ billion in volume. What powers this:

Multi-Chain Support: Ethereum, Arbitrum, Base, Solana. Users trade anywhere from one interface.

Token Sniping: Automatic purchase when new tokens list. Speed is everything - millisecond advantages matter.

MEV Protection: Private transactions prevent frontrunning. Sandwich attacks can't see your trades.

Copy Trading: Follow whale wallets automatically. When they buy, you buy. Democratizes alpha.

Revenue Sharing: UNIBOT token holders share fees. Aligns user and platform incentives.

The technical requirements:

class TradingBotArchitecture:

```
def __init__(self):
    self.components = {
        "telegram_interface": {
            "purpose": "User commands and notifications",
            "tech": "python-telegram-bot or aiogram",
            "scale": "Thousands of concurrent users"
        },
    },
```

```
"chain_monitors": {
  "purpose": "Watch for new tokens, price changes, whale moves",
  "tech": "WebSocket connections to RPC nodes",
  "scale": "Multiple chains, real-time"
},
"execution_engine": {
  "purpose": "Submit transactions with optimal gas",
  "tech": "Direct node connections, flashbots for MEV protection",
  "scale": "Millisecond execution"
},
"user_database": {
  "purpose": "Wallets, settings, positions, history",
  "tech": "PostgreSQL with Redis cache",
  "scale": "Millions of records"
},
"price_aggregator": {
  "purpose": "Best prices across DEXs",
  "tech": "DEX aggregator APIs, direct pool queries",
  "scale": "Sub-second updates"
},
"risk_engine": {
  "purpose": "Position limits, rug detection, slippage protection",
  "tech": "Custom rules engine",
  "scale": "Every trade validated"
}
}
```

Maestro Patterns:

600,000 users across 10+ chains. Key features:

Whale Tracking: Monitor large wallet movements. Alert users to potential alpha.

Anti-Rug Detection: Analyze contracts for red flags. Honeypots, tax manipulation, ownership risks.

Flexible Pricing: Free tier with 1% fee or \$200/month premium. Captures different user segments.


```
class WhaleTracker:
```

```
    def __init__(self, min_value_usd = 100000):  
        self.min_value_usd = min_value_usd  
        self.tracked_wallets = set()  
        self.subscribers = {}
```

```
    async def add_whale_wallet(self, address, label = None):  
        self.tracked_wallets.add(address)
```

```
    async def subscribe_user(self, user_id, wallet_address):  
        if wallet_address not in self.subscribers:  
            self.subscribers[wallet_address] = set()  
            self.subscribers[wallet_address].add(user_id)
```

```
    async def process_transaction(self, tx):  
        from_addr = tx["from"]  
        to_addr = tx["to"]  
        value_usd = tx["value_usd"]
```

```
        if value_usd < self.min_value_usd:  
            return
```

```
        if from_addr in self.tracked_wallets:  
            await self.notify_whale_movement(from_addr, tx, "sell")
```

```
        if to_addr in self.tracked_wallets:  
            await self.notify_whale_movement(to_addr, tx, "buy")
```

```
    async def notify_whale_movement(self, whale_address, tx, action):  
        subscribers = self.subscribers.get(whale_address, set())
```

```
        for user_id in subscribers:  
            await self.send_alert(user_id, whale_address, tx, action)
```

```
    async def send_alert(self, user_id, whale, tx, action):  
        message = (  
            f"Whale Alert!\n\n"  
            f"Wallet: {whale[:8]}...{whale[-6:]} \n")
```

```
f'Action: {action.upper()}\n"
f'Token: {tx['token_symbol']}\n"
f'Amount: ${tx['value_usd'];:.0f}\n"
f'TX: {tx['hash'][:16]}..."
)

print(f'Alert to {user_id}: {message}')
```

Banana Gun Approach:

Focus on speed and specific chains. Auto sniping, copy trading, MEV resistance. Revenue sharing through token.

```
class SnipingEngine:

    def __init__(self):
        self.pending_snipes = {}
        self.max_slippage = 0.10
        self.max_gas_multiplier = 3.0

    async def setup_snipe(
        self,
        user_id,
        token_address,
        amount_eth,
        max_price=None,
        auto_sell_multiplier=None
    ):
        snipe_config = {
            "user_id": user_id,
            "token": token_address,
            "amount": amount_eth,
            "max_price": max_price,
            "auto_sell": auto_sell_multiplier,
            "status": "pending",
            "created": time.time()
        }

        snipe_id = f'{user_id}:{token_address}:{time.time()}'
```

```
self.pending_snipes[snipe_id] = snipe_config
```

```
return snipe_id
```

```
async def execute_snipe(self, snipe_id, pool_address):
```

```
    config = self.pending_snipes.get(snipe_id)
```

```
    if not config:
```

```
        return None
```

```
    current_price = await self.get_token_price(config["token"])
```

```
    if config["max_price"] and current_price > config["max_price"]:
```

```
        return {"status": "skipped", "reason": "price_exceeded"}
```

```
    tx_hash = await self.submit_swap(
```

```
        token_in="ETH",
```

```
        token_out=config["token"],
```

```
        amount_in=config["amount"],
```

```
        pool=pool_address
```

```
    )
```

```
    config["status"] = "executed"
```

```
    config["tx_hash"] = tx_hash
```

```
    if config["auto_sell"]:
```

```
        await self.setup_auto_sell(
```

```
            config["user_id"],
```

```
            config["token"],
```

```
            config["auto_sell"]
```

```
        )
```

```
    return {"status": "success", "tx_hash": tx_hash}
```

```
async def get_token_price(self, token):
```

```
    return 0.0001
```

```
async def submit_swap(self, token_in, token_out, amount_in, pool):
```

```
    return "0x" + "a" * 64
```

```
async def setup_auto_sell(self, user_id, token, multiplier):
```

```
print(f'Auto-sell setup: {token} at {multiplier}x for user {user_id}')
```

SECTION 7: COMMUNITY BOT PATTERNS

Crypto projects live and die by their communities. Telegram groups need automation to function at scale.

Token-Gated Access:

Only token holders can join. Verifies ownership, grants access, removes when sold.

```
class TokenGatedGroup:
```

```
    def __init__(self, bot_token, required_token, min_balance):
        self.bot = Bot(bot_token)
        self.required_token = required_token
        self.min_balance = min_balance
        self.verified_users = {}
```

```
    async def verify_and_grant_access(self, user_id, wallet_address,
group_id):
        balance = await self.check_token_balance(wallet_address)
```

```
        if balance < self.min_balance:
            return {
                "success": False,
                "reason": f'Insufficient balance. Need {self.min_balance}, have
{balance}'
            }
```

```
        self.verified_users[user_id] = {
            "wallet": wallet_address,
            "verified_balance": balance,
            "verified_at": time.time()
        }
```

```
        invite_link = await self.bot.create_chat_invite_link(
            chat_id=group_id,
```

```
member_limit = 1,  
expire_date = int(time.time()) + 3600  
)
```

```
return {  
    "success": True,  
    "invite_link": invite_link.invite_link  
}
```

```
async def check_token_balance(self, wallet_address):  
    return 1000
```

```
async def periodic_verification(self, group_id):  
    for user_id, data in list(self.verified_users.items()):  
        current_balance = await self.check_token_balance(data["wallet"])
```

```
        if current_balance < self.min_balance:  
            await self.remove_user(group_id, user_id)  
            del self.verified_users[user_id]
```

```
    async def remove_user(self, group_id, user_id):  
        try:  
            await self.bot.ban_chat_member(  
                chat_id = group_id,  
                user_id = user_id  
            )
```

```
            await self.bot.unban_chat_member(  
                chat_id = group_id,  
                user_id = user_id  
            )
```

```
            await self.bot.send_message(  
                chat_id = user_id,  
                text = "Your token balance dropped below the required amount. "  
                "You've been removed from the group. "  
                "Buy more tokens to rejoin."  
            )  
        except Exception as e:  
            print(f'Failed to remove user {user_id}: {e}')
```

Anti-Spam Protection:

Large groups attract spam. Automated protection is essential.

```
class AntiSpamBot:
```

```
    def __init__(self, bot_token):
```

```
        self.bot = Bot(bot_token)
```

```
        self.user_messages = {}
```

```
        self.spam_patterns = [
```

```
            r"t\.me/\w+",
```

```
            r"join.*airdrop",
```

```
            r"free.*tokens",
```

```
            r"guaranteed.*profit",
```

```
            r"dm.*for.*details"
```

```
        ]
```

```
        self.rate_limit = 5
```

```
        self.rate_window = 60
```

```
    async def check_message(self, update: Update):
```

```
        user_id = update.effective_user.id
```

```
        message = update.effective_message
```

```
        text = message.text or message.caption or ""
```

```
        if await self.is_spam_content(text):
```

```
            await self.handle_spam(message, "spam_content")
```

```
            return True
```

```
        if await self.is_rate_limited(user_id):
```

```
            await self.handle_spam(message, "rate_limit")
```

```
            return True
```

```
        self.record_message(user_id)
```

```
        return False
```

```
    async def is_spam_content(self, text):
```

```
        import re
```

```
text_lower = text.lower()
```

```
for pattern in self.spam_patterns:  
    if re.search(pattern, text_lower):  
        return True
```

```
return False
```

```
async def is_rate_limited(self, user_id):  
    now = time.time()  
    user_history = self.user_messages.get(user_id, [])
```

```
    recent = [ts for ts in user_history if now - ts < self.rate_window]  
    self.user_messages[user_id] = recent
```

```
    return len(recent) >= self.rate_limit
```

```
def record_message(self, user_id):  
    if user_id not in self.user_messages:  
        self.user_messages[user_id] = []  
    self.user_messages[user_id].append(time.time())
```

```
async def handle_spam(self, message, reason):  
    try:  
        await message.delete()  
    except Exception:  
        pass
```

```
    user_id = message.from_user.id  
    chat_id = message.chat.id
```

```
    if reason == "spam_content":  
        await self.bot.ban_chat_member(  
            chat_id=chat_id,  
            user_id=user_id  
        )  
    elif reason == "rate_limit":  
        await self.bot.restrict_chat_member(  
            chat_id=chat_id,  
            user_id=user_id,
```

```
until_date=int(time.time()) + 300,  
permissions={"can_send_messages": False}  
)
```

SECTION 8: DEVELOPMENT ENVIRONMENT SETUP

Setting up a proper development environment for Telegram bot development.

Project Structure:

```
crypto-telegram-bot/  
  config/  
  settings.py  
  logging.py  
  handlers/  
  commands.py  
  callbacks.py  
  payments.py  
  services/  
  blockchain.py  
  database.py  
  cache.py  
  mini_app/  
  public/  
  src/  
  package.json  
  tests/  
  test_handlers.py  
  test_services.py  
  main.py  
  requirements.txt  
  docker-compose.yml
```

Python Dependencies (requirements.txt):

```
python-telegram-bot = 22.5  
aiogram = 3.4.1
```



```
aiohttp = = 3.9.3
asyncpg = = 0.29.0
redis = = 5.0.1
sqlalchemy = = 2.0.25
alembic = = 1.13.1
pydantic = = 2.6.0
python-dotenv = = 1.0.1
structlog = = 24.1.0
pytest = = 8.0.0
pytest-asyncio = = 0.23.4
httpx = = 0.26.0
```

Environment Configuration:

```
import os
from pydantic import BaseSettings
from functools import lru_cache

class Settings(BaseSettings):
    bot_token: str
    webhook_url: str = ""
    webhook_secret: str = ""

    database_url: str
    redis_url: str = "redis://localhost:6379"

    ton_rpc_url: str = "https://toncenter.com/api/v2"
    ton_api_key: str = ""

    fragment_api_key: str = ""

    log_level: str = "INFO"
    environment: str = "development"

    min_stars_withdrawal: int = 1000
    star_to_ton_rate: float = 0.01

class Config:
    env_file = ".env"
```

```
@lru_cache()
def get_settings():
    return Settings()

settings = get_settings()
```

Docker Development Environment:

```
version: '3.8'
```

```
services:
```

```
  bot:
```

```
    build: .
```

```
    volumes:
```

```
      - ./app
```

```
    environment:
```

```
      - BOT_TOKEN=${BOT_TOKEN}
```

```
      - DATABASE_URL=postgresql://postgres:postgres@db:5432/botdb
```

```
      - REDIS_URL=redis://redis:6379
```

```
    depends_on:
```

```
      - db
```

```
      - redis
```

```
  db:
```

```
    image: postgres:15
```

```
    environment:
```

```
      POSTGRES_DB: botdb
```

```
      POSTGRES_USER: postgres
```

```
      POSTGRES_PASSWORD: postgres
```

```
    volumes:
```

```
      - postgres_data:/var/lib/postgresql/data
```

```
    ports:
```

```
      - "5432:5432"
```

```
  redis:
```

```
    image: redis:7-alpine
```

```
    ports:
```

```
      - "6379:6379"
```

```
mini_app:
build: ./mini_app
ports:
- "3000:3000"
volumes:
- ./mini_app:/app
- /app/node_modules
```

```
volumes:
postgres_data:
```

SECTION 9: TESTING TELEGRAM BOTS

Testing bot logic without hitting real APIs.

```
from unittest.mock import AsyncMock, MagicMock, patch
import pytest
from telegram import Update, User, Chat, Message
```

```
class TelegramTestFactory:
```

```
    @staticmethod
    def create_user(user_id = 12345, username = "testuser",
first_name = "Test"):
        return User(
            id = user_id,
            first_name = first_name,
            is_bot = False,
            username = username
        )
```

```
    @staticmethod
    def create_chat(chat_id = 12345, chat_type = "private"):
        return Chat(id = chat_id, type = chat_type)
```

```
    @staticmethod
    def create_message(
        text,
```

```
user_id = 12345,  
chat_id = 12345,  
message_id = 1  
):  
user = TelegramTestFactory.create_user(user_id)  
chat = TelegramTestFactory.create_chat(chat_id)
```

```
message = MagicMock(spec=Message)  
message.message_id = message_id  
message.text = text  
message.from_user = user  
message.chat = chat  
message.reply_text = AsyncMock()  
message.delete = AsyncMock()
```

```
return message
```

```
@staticmethod  
def create_update(  
text,  
user_id = 12345,  
chat_id = 12345  
):  
message = TelegramTestFactory.create_message(  
text, user_id, chat_id  
)
```

```
update = MagicMock(spec=Update)  
update.message = message  
update.effective_user = message.from_user  
update.effective_chat = message.chat  
update.effective_message = message  
update.callback_query = None
```

```
return update
```

```
@staticmethod  
def create_context(args=None, user_data=None):  
context = MagicMock()  
context.args = args or []
```

```
context.user_data = user_data or {}
context.bot = AsyncMock()
return context
```

```
class TestPriceCommand:
```

```
@pytest.fixture
def price_bot(self):
    from your_bot import PriceBot
    return PriceBot("fake_token")
```

```
@pytest.mark.asyncio
async def test_price_command_with_valid_symbol(self, price_bot):
    update = TelegramTestFactory.create_update("/price ETH")
    context = TelegramTestFactory.create_context(args=["ETH"])
```

```
    with patch.object(price_bot, 'get_price', return_value=2345.67):
        await price_bot.handle_price(update, context)
```

```
        update.message.reply_text.assert_called_once()
        call_args = update.message.reply_text.call_args[0][0]
        assert "ETH" in call_args
        assert "2345.67" in call_args or "2,345.67" in call_args
```

```
@pytest.mark.asyncio
async def test_price_command_without_symbol(self, price_bot):
    update = TelegramTestFactory.create_update("/price")
    context = TelegramTestFactory.create_context(args=[])
```

```
    await price_bot.handle_price(update, context)
```

```
    call_args = update.message.reply_text.call_args[0][0]
    assert "usage" in call_args.lower() or "specify" in call_args.lower()
```

```
class TestPaymentFlow:
```

```
@pytest.mark.asyncio
async def test_successful_payment_processing(self):
```

```
from your_bot import PaymentBot
```

```
bot = PaymentBot("fake_token")
```

```
update = MagicMock()
```

```
update.message.successful_payment.invoice_payload =  
"premium:monthly"
```

```
update.message.successful_payment.total_amount = 100
```

```
update.message.successful_payment.telegram_payment_charge_id =  
"charge_123"
```

```
update.effective_user.id = 12345
```

```
update.message.reply_text = AsyncMock()
```

```
with patch.object(bot, 'activate_premium',  
new_callable=AsyncMock) as mock_activate:  
    await bot.handle_successful_payment(update, None)
```

```
mock_activate.assert_called_once_with(12345, "monthly")
```

SECTION 10: NEXT STEPS

This chapter mapped the Telegram ecosystem. You now understand:

The layered architecture - Telegram platform, TON blockchain, your application, and external systems. Every feature touches multiple layers.

Bot API 8.x capabilities - message streaming, Business 2.0, reactions, and the upcoming 64-bit user ID requirement.

Mini Apps 2.0 - full-screen mode, device sensors, storage APIs, and subscription payments. Real applications, not just web views.

Telegram Stars - mandatory for digital goods, 21-day hold period, withdrawal through Fragment to TON. The economics of monetization.

TON exclusivity - all Telegram blockchain features use TON. TON Connect is the only wallet protocol. This isn't changing.

Trading bot patterns - what makes Trojan, Maestro, and Banana Gun successful. The technical requirements for billion-dollar volume.

Community automation - token-gating, anti-spam, whale alerts. Managing groups at scale.

Development setup - project structure, dependencies, Docker environment, testing patterns.

Next chapter: Bot Development Fundamentals. We build complete bots with python-telegram-bot v22 and aiogram.

CHAPTER SUMMARY

Telegram in 2025 is a complete crypto operating system:

One billion users provide the audience
TON blockchain provides exclusive infrastructure
Telegram Stars provide mandatory payment rails
Mini Apps 2.0 provide application capabilities
Trading bots prove the market - 50+ billion in volume

Building here means building on TON, accepting Stars, and designing for scale from day one.

The opportunity is massive. The competition is real. Let's build.

CHAPTER 2: BOT DEVELOPMENT FUNDAMENTALS

Building production-ready Telegram bots with modern Python libraries. This chapter covers python-telegram-bot v22 and aiogram 3.x - async-first architectures that handle thousands of concurrent users. You'll understand handlers, conversations, keyboards, and error recovery patterns.

SECTION 1: BOT CREATION AND SETUP

Every Telegram bot starts with BotFather. This official bot manages bot creation, tokens, commands, and settings.

Creating a Bot:

1. Open Telegram and search for @BotFather
2. Send /newbot
3. Provide a display name (e.g., "Crypto Trading Bot")
4. Provide a username ending in "bot" (e.g., "my_crypto_trading_bot")
5. Receive your token (format:
123456789:ABCdefGHIjklMNOpqrsTUVwxyz)

The token is your bot's identity and authentication. Anyone with this token controls your bot completely. Store it securely. Never commit it to git. Use environment variables.

Configuring Bot Settings via BotFather:

/setdescription - Short description shown when users start chat
/setabouttext - About section in bot profile
/setuserpic - Bot profile picture
/setcommands - Command list for menu button
/setprivacy - Whether bot sees all group messages (default: only commands)
/mybots - Manage all your bots

Setting Commands:

Send /setcommands to BotFather, select your bot, then provide commands in this format:

start - Start the bot and see welcome message
help - View available commands
price - Get current crypto prices
wallet - View your wallet balance
trade - Execute a trade
alerts - Manage price alerts
settings - Configure your preferences

These commands appear in the menu button (/) in Telegram. Users can tap to insert commands without typing.

Project Setup:

Create a new Python project with virtual environment:

```
python -m venv venv
source venv/bin/activate # Linux/Mac
venv\Scripts\activate # Windows
```

```
pip install python-telegram-bot == 22.5
pip install python-dotenv
pip install asyncpg
pip install redis
pip install structlog
```

Create project structure:

```
crypto_bot/
__init__.py
main.py
config.py
handlers/
__init__.py
commands.py
callbacks.py
conversations.py
services/
__init__.py
price.py
wallet.py
database.py
utils/
__init__.py
decorators.py
formatters.py
tests/
__init__.py
test_handlers.py
```

```
.env
requirements.txt
```

Configuration Module (config.py):

```
import os
from dataclasses import dataclass
from dotenv import load_dotenv

load_dotenv()

@dataclass
class Config:
    bot_token: str
    database_url: str
    redis_url: str
    admin_user_ids: list
    environment: str

    @classmethod
    def from_env(cls):
        admin_ids_str = os.getenv("ADMIN_USER_IDS", "")
        admin_ids = [int(x) for x in admin_ids_str.split(",") if x]

        return cls(
            bot_token=os.getenv("BOT_TOKEN"),
            database_url=os.getenv("DATABASE_URL"),
            redis_url=os.getenv("REDIS_URL", "redis://localhost:6379"),
            admin_user_ids=admin_ids,
            environment=os.getenv("ENVIRONMENT", "development")
        )

config = Config.from_env()
```

SECTION 2: PYTHON-TELEGRAM-BOT V22 ARCHITECTURE

The python-telegram-bot library uses an Application-centered architecture. Understanding the components helps you structure

code correctly.

Core Components:

Application: The main entry point. Manages updater, handlers, job queue, and bot instance.

Updater: Fetches updates from Telegram (polling) or receives them (webhook). Runs in background.

Dispatcher: Routes updates to handlers based on filters and type.

Handlers: Process specific update types. `CommandHandler` for / commands, `MessageHandler` for text, `CallbackQueryHandler` for button clicks.

Context: Carries data through the handler chain. `user_data`, `chat_data`, and `bot_data` persist across calls.

Basic Bot Structure:

```
import asyncio
import logging
from telegram import Update
from telegram.ext import (
    Application,
    CommandHandler,
    MessageHandler,
    CallbackQueryHandler,
    filters,
    ContextTypes
)
from config import config

logging.basicConfig(
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    level=logging.INFO
)
logger = logging.getLogger(__name__)
```

```
class CryptoBot:
```

```
    def __init__(self):
        self.application = (
            Application.builder()
            .token(config.bot_token)
            .build()
        )
        self.setup_handlers()
```

```
    def setup_handlers(self):
        self.application.add_handler(
            CommandHandler("start", self.cmd_start)
        )
        self.application.add_handler(
            CommandHandler("help", self.cmd_help)
        )
        self.application.add_handler(
            CommandHandler("price", self.cmd_price)
        )
        self.application.add_handler(
            MessageHandler(
                filters.TEXT & ~filters.COMMAND,
                self.on_text_message
            )
        )
        self.application.add_handler(
            CallbackQueryHandler(self.on_callback)
        )
        self.application.add_error_handler(self.on_error)
```

```
    async def cmd_start(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
        user = update.effective_user
```

```
        context.user_data["registered"] = True
        context.user_data["username"] = user.username
        context.user_data["first_seen"] = update.message.date.isoformat()
```

```
    welcome_text = (
```

```
f"Welcome to CryptoBot, {user.first_name}!\n\n"
f"I help you track prices, manage wallets, and trade crypto.\n\n"
f"Quick start:\n"
f"/price ETH - Check ETH price\n"
f"/wallet - View your wallet\n"
f"/help - See all commands"
)
```

```
await update.message.reply_text(welcome_text)
```

```
async def cmd_help(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
```

```
    help_text = (
        "<b>CryptoBot Commands</b> \n\n"
        "<b>Prices</b> \n"
        "/price [symbol] - Get current price\n"
        "/chart [symbol] - View price chart\n"
        "/alerts - Manage price alerts\n\n"
        "<b>Wallet</b> \n"
        "/wallet - View your wallet\n"
        "/deposit - Get deposit address\n"
        "/withdraw - Withdraw funds\n\n"
        "<b>Trading</b> \n"
        "/trade - Open trade menu\n"
        "/positions - View open positions\n"
        "/history - Trade history\n\n"
        "<b>Settings</b> \n"
        "/settings - Configure preferences\n"
        "/notifications - Alert settings"
    )
```

```
await update.message.reply_text(help_text, parse_mode="HTML")
```

```
async def cmd_price(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
```

```
    args = context.args
```

```
    if not args:
```

```
        await update.message.reply_text(
            "Usage: /price [symbol]\n"
```

```
"Example: /price ETH"
```

```
)
```

```
return
```

```
symbol = args[0].upper()
```

```
price_data = await self.fetch_price(symbol)
```

```
if price_data is None:
```

```
    await update.message.reply_text(
```

```
        f'Could not find price for {symbol}')
```

```
)
```

```
return
```

```
response = (
```

```
    f'<b>{symbol}</b>\n\n'
```

```
    f'Price: ${price_data['price']:,.2f}\n'
```

```
    f'24h Change: {price_data['change_24h']: +.2f}%\n'
```

```
    f'Volume: ${price_data['volume']:,.0f}')
```

```
)
```

```
await update.message.reply_text(response, parse_mode="HTML")
```

```
async def fetch_price(self, symbol):
```

```
    prices = {
```

```
        "BTC": {"price": 43210.50, "change_24h": 2.3, "volume":
```

```
        28_000_000_000},
```

```
        "ETH": {"price": 2345.67, "change_24h": -1.2, "volume":
```

```
        15_000_000_000},
```

```
        "TON": {"price": 5.67, "change_24h": 5.8, "volume": 500_000_000},
```

```
        "SOL": {"price": 98.76, "change_24h": 3.4, "volume":
```

```
        2_000_000_000}
```

```
    }
```

```
    return prices.get(symbol)
```

```
async def on_text_message(self, update: Update, context:
```

```
ContextTypes.DEFAULT_TYPE):
```

```
    text = update.message.text.lower()
```

```
    keywords = {
```

```
"price": "Use /price [symbol] to check prices",
"help": "Use /help to see all commands",
"wallet": "Use /wallet to view your wallet",
"buy": "Use /trade to start trading",
"sell": "Use /trade to start trading"
}
```

```
for keyword, response in keywords.items():
    if keyword in text:
        await update.message.reply_text(response)
    return
```

```
await update.message.reply_text(
    "I didn't understand that. Use /help to see available commands."
)
```

```
async def on_callback(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
    query = update.callback_query
    await query.answer()
```

```
data = query.data
```

```
logger.info(f"Callback received: {data}")
```

```
async def on_error(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
    logger.error(f"Exception while handling update: {context.error}")
```

```
if update and update.effective_message:
    await update.effective_message.reply_text(
        "An error occurred. Please try again later."
    )
```

```
def run(self):
    logger.info("Starting bot...")
    self.application.run_polling(
        allowed_updates=Update.ALL_TYPES,
        drop_pending_updates=True
    )
```

```
if __name__ == "__main__":  
    bot = CryptoBot()  
    bot.run()
```

SECTION 3: AIOGRAM 3.X ALTERNATIVE

aiogram is another popular async Telegram library. It uses a dispatcher pattern similar to web frameworks. Some developers prefer its middleware system and explicit routing.

aiogram Basic Setup:

```
import asyncio  
import logging  
from aiogram import Bot, Dispatcher, Router, types, F  
from aiogram.filters import Command, CommandStart  
from aiogram.enums import ParseMode  
from config import config
```

```
logging.basicConfig(level=logging.INFO)  
logger = logging.getLogger(__name__)
```

```
router = Router()
```

```
class AiogramCryptoBot:
```

```
    def __init__(self):  
        self.bot = Bot(  
            token=config.bot_token,  
            default=types.DefaultBotProperties(  
                parse_mode=ParseMode.HTML  
            )  
        )  
        self.dp = Dispatcher()  
        self.dp.include_router(router)
```

```
    async def run(self):  
        logger.info("Starting aiogram bot...")
```



```
await self.dp.start_polling(self.bot)
```

```
@router.message(CommandStart())
```

```
async def cmd_start(message: types.Message):
```

```
    user = message.from_user
```

```
    await message.answer(
```

```
        f'Welcome to CryptoBot, {user.first_name}!\n\n'
```

```
        f'Use /help to see available commands.'
```

```
    )
```

```
@router.message(Command("help"))
```

```
async def cmd_help(message: types.Message):
```

```
    await message.answer(
```

```
        "<b>Commands</b>\n\n"
```

```
        "/price [symbol] - Get price\n"
```

```
        "/wallet - View wallet\n"
```

```
        "/trade - Start trading"
```

```
    )
```

```
@router.message(Command("price"))
```

```
async def cmd_price(message: types.Message):
```

```
    args = message.text.split()[1:] if len(message.text.split()) > 1 else  
    []
```

```
    if not args:
```

```
        await message.answer("Usage: /price ETH")
```

```
        return
```

```
    symbol = args[0].upper()
```

```
    await message.answer(f'<b>{symbol}</b>: $2,345.67')
```

```
@router.message(F.text & ~F.text.startswith("/"))
```

```
async def on_text(message: types.Message):
```

```
    await message.answer(
```

```
        "Use /help to see commands."
```

```
    )
```

```
if __name__ == "__main__":
```

```
    bot = AiogramCryptoBot()
```

```
asyncio.run(bot.run())
```

aiogram Middleware:

aiogram's middleware system enables cross-cutting concerns like logging, authentication, and rate limiting.

```
from aiogram import BaseMiddleware
from aiogram.types import TelegramObject, Update
from typing import Callable, Dict, Any, Awaitable
import time
```

```
class LoggingMiddleware(BaseMiddleware):
```

```
    async def _call_(
        self,
        handler: Callable[[TelegramObject, Dict[str, Any]],
        Awaitable[Any]],
        event: TelegramObject,
        data: Dict[str, Any]
    ) -> Any:
        start_time = time.time()
```

```
        if isinstance(event, Update) and event.message:
```

```
            user_id = event.message.from_user.id
```

```
            text = event.message.text or "[non-text]"
```

```
            logger.info(f"User {user_id}: {text[:50]}")
```

```
        result = await handler(event, data)
```

```
        duration = time.time() - start_time
```

```
        logger.info(f"Handler completed in {duration:.3f}s")
```

```
        return result
```

```
class AuthMiddleware(BaseMiddleware):
```

```
    def __init__(self, required_user_ids: list = None):
```

```
self.required_user_ids = required_user_ids or []
```

```
async def _call_(
    self,
    handler: Callable[[TelegramObject, Dict[str, Any]],
Awaitable[Any]],
    event: TelegramObject,
    data: Dict[str, Any]
) -> Any:
    if isinstance(event, Update) and event.message:
        user_id = event.message.from_user.id

        if self.required_user_ids and user_id not in self.required_user_ids:
            await event.message.answer("Unauthorized.")
            return

    return await handler(event, data)
```

```
class RateLimitMiddleware(BaseMiddleware):
```

```
    def __init__(self, limit: int = 30, window: int = 60):
        self.limit = limit
        self.window = window
        self.user_requests = {}
```

```
    async def _call_(
        self,
        handler: Callable[[TelegramObject, Dict[str, Any]],
Awaitable[Any]],
        event: TelegramObject,
        data: Dict[str, Any]
    ) -> Any:
        if isinstance(event, Update) and event.message:
            user_id = event.message.from_user.id
            now = time.time()

            if user_id not in self.user_requests:
                self.user_requests[user_id] = []
```

```

self.user_requests[user_id] = [
    ts for ts in self.user_requests[user_id]
    if now - ts < self.window
]

if len(self.user_requests[user_id]) >= self.limit:
    await event.message.answer(
        "Rate limited. Please wait before sending more messages."
    )
return

self.user_requests[user_id].append(now)

return await handler(event, data)

```

SECTION 4: INLINE KEYBOARDS AND CALLBACKS

Inline keyboards create interactive interfaces within messages. Buttons trigger callback queries that your bot handles.

Building Keyboard Layouts:

```

from telegram import InlineKeyboardButton,
InlineKeyboardMarkup

class KeyboardBuilder:

    @staticmethod
    def main_menu():
        keyboard = [
            [
                InlineKeyboardButton("Wallet", callback_data="menu:wallet"),
                InlineKeyboardButton("Trade", callback_data="menu:trade")
            ],
            [
                InlineKeyboardButton("Prices", callback_data="menu:prices"),
                InlineKeyboardButton("Alerts", callback_data="menu:alerts")
            ],
            [

```

```

InlineKeyboardButton("Settings", callback_data="menu:settings")
]
]
return InlineKeyboardMarkup(keyboard)

```

```

@staticmethod
def trade_menu():
    keyboard = [
        [
            InlineKeyboardButton("Buy", callback_data="trade:buy"),
            InlineKeyboardButton("Sell", callback_data="trade:sell")
        ],
        [
            InlineKeyboardButton("Swap", callback_data="trade:swap")
        ],
        [
            InlineKeyboardButton("Back", callback_data="menu:main")
        ]
    ]
    return InlineKeyboardMarkup(keyboard)

```

```

@staticmethod
def token_selector(tokens, action):
    keyboard = []
    row = []

```

```

    for token in tokens:
        row.append(
            InlineKeyboardButton(
                token,
                callback_data=f'{action}:{token}'
            )
        )

```

```

    if len(row) == 3:
        keyboard.append(row)
        row = []

```

```

    if row:
        keyboard.append(row)

```

```
keyboard.append([
InlineKeyboardButton("Cancel", callback_data = "menu:main")
])
```

```
return InlineKeyboardMarkup(keyboard)
```

```
@staticmethod
```

```
def amount_selector(token, action):
```

```
keyboard = [
```

```
[
```

```
InlineKeyboardButton("25%", callback_data = f'amount:{action}:
{token}:25"),
```

```
InlineKeyboardButton("50%", callback_data = f'amount:{action}:
{token}:50"),
```

```
InlineKeyboardButton("75%", callback_data = f'amount:{action}:
{token}:75"),
```

```
InlineKeyboardButton("100%", callback_data = f'amount:{action}:
{token}:100")
```

```
],
```

```
[
```

```
InlineKeyboardButton("Custom Amount", callback_data = f'amount:
{action}:{token}:custom")
```

```
],
```

```
[
```

```
InlineKeyboardButton("Back", callback_data = "trade:menu")
```

```
]
```

```
]
```

```
return InlineKeyboardMarkup(keyboard)
```

```
@staticmethod
```

```
def confirmation(action_data):
```

```
keyboard = [
```

```
[
```

```
InlineKeyboardButton("Confirm", callback_data = f'confirm:
{action_data}"),
```

```
InlineKeyboardButton("Cancel", callback_data = "cancel")
```

```
]
```

```
]
```

```
return InlineKeyboardMarkup(keyboard)
```

Handling Callbacks:

```
class CallbackHandler:
```

```
    def __init__(self, bot_instance):  
        self.bot = bot_instance
```

```
    async def handle_callback(self, update: Update, context:  
ContextTypes.DEFAULT_TYPE):  
        query = update.callback_query  
        await query.answer()
```

```
        data = query.data  
        parts = data.split(":")
```

```
        handler_map = {  
            "menu": self.handle_menu,  
            "trade": self.handle_trade,  
            "amount": self.handle_amount,  
            "confirm": self.handle_confirm,  
            "cancel": self.handle_cancel  
        }
```

```
        handler = handler_map.get(parts[0])
```

```
        if handler:  
            await handler(query, parts[1:], context)  
        else:  
            await query.message.edit_text("Unknown action")
```

```
    async def handle_menu(self, query, parts, context):  
        action = parts[0] if parts else "main"
```

```
        menu_handlers = {  
            "main": self.show_main_menu,  
            "wallet": self.show_wallet,  
            "trade": self.show_trade_menu,  
            "prices": self.show_prices,
```

```
"alerts": self.show_alerts,  
"settings": self.show_settings  
}
```

```
handler = menu_handlers.get(action)
```

```
if handler:  
    await handler(query, context)
```

```
async def show_main_menu(self, query, context):  
    await query.message.edit_text(  
        "Main Menu\n\n"  
        "Select an option:",  
        reply_markup=KeyboardBuilder.main_menu()  
    )
```

```
async def show_wallet(self, query, context):  
    wallet_text = (  
        "<b>Your Wallet</b>\n\n"  
        "ETH: 1.2345\n"  
        "USDT: 5,000.00\n"  
        "TON: 500.00\n\n"  
        "Total: ~$8,234.56"  
    )
```

```
keyboard = InlineKeyboardMarkup(  
    [  
        InlineKeyboardButton("Deposit", callback_data="wallet:deposit"),  
        InlineKeyboardButton("Withdraw",  
            callback_data="wallet:withdraw")  
    ],  
    [InlineKeyboardButton("Back", callback_data="menu:main")]  
)
```

```
await query.message.edit_text(  
    wallet_text,  
    reply_markup=keyboard,  
    parse_mode="HTML"  
)
```



```
async def show_trade_menu(self, query, context):
    await query.message.edit_text(
        "Trade Menu\n\n"
        "Select action:",
        reply_markup=KeyboardBuilder.trade_menu()
    )
```

```
async def show_prices(self, query, context):
    prices_text = (
        "<b> Current Prices </b> \n\n"
        "BTC: $43,210.50 (+ 2.3%) \n"
        "ETH: $2,345.67 (-1.2%) \n"
        "TON: $5.67 (+ 5.8%) \n"
        "SOL: $98.76 (+ 3.4%) \n\n"
        "Updated: just now"
    )
```

```
keyboard = InlineKeyboardMarkup([
    [InlineKeyboardButton("Refresh", callback_data="menu:prices")],
    [InlineKeyboardButton("Back", callback_data="menu:main")]
])
```

```
await query.message.edit_text(
    prices_text,
    reply_markup=keyboard,
    parse_mode="HTML"
)
```

```
async def show_alerts(self, query, context):
    await query.message.edit_text(
        "Price Alerts\n\n"
        "No active alerts. \n\n"
        "Tap below to create one.",
        reply_markup=InlineKeyboardMarkup([
            [InlineKeyboardButton("Create Alert",
                callback_data="alerts:create")],
            [InlineKeyboardButton("Back", callback_data="menu:main")]
        ])
    )
```

```

async def show_settings(self, query, context):
    notifications = context.user_data.get("notifications", True)
    currency = context.user_data.get("currency", "USD")

    await query.message.edit_text(
        "<b>Settings</b>\n\n"
        f'Notifications: {'ON' if notifications else 'OFF'}\n'
        f'Currency: {currency}',
        reply_markup=InlineKeyboardMarkup([
            [InlineKeyboardButton(
                f'Notifications: {'ON' if notifications else 'OFF'}',
                callback_data="settings:toggle_notifications"
            )],
            [InlineKeyboardButton(
                f'Currency: {currency}',
                callback_data="settings:currency"
            )],
            [InlineKeyboardButton("Back", callback_data="menu:main")]
        ]),
        parse_mode="HTML"
    )

```

```

async def handle_trade(self, query, parts, context):
    action = parts[0] if parts else "menu"

    if action == "menu":
        await self.show_trade_menu(query, context)
    elif action == "buy":
        await query.message.edit_text(
            "Select token to buy:",
            reply_markup=KeyboardBuilder.token_selector(
                ["ETH", "BTC", "TON", "SOL", "USDT"],
                "buy"
            )
        )
    elif action == "sell":
        await query.message.edit_text(
            "Select token to sell:",
            reply_markup=KeyboardBuilder.token_selector(
                ["ETH", "BTC", "TON"],

```

```
"sell"  
)  
)
```

```
async def handle_amount(self, query, parts, context):  
    action = parts[0]  
    token = parts[1]  
    percentage = parts[2]
```

```
    if percentage == "custom":  
        context.user_data["awaiting_amount"] = {  
            "action": action,  
            "token": token  
        }  
        await query.message.edit_text(  
            f"Enter amount of {token} to {action}:"  
        )  
        return
```

```
        context.user_data["pending_trade"] = {  
            "action": action,  
            "token": token,  
            "percentage": int(percentage)  
        }
```

```
        await query.message.edit_text(  
            f"<b> Confirm {action.upper()} </b>\n\n"  
            f"Token: {token}\n"  
            f"Amount: {percentage}% of balance\n"  
            f"Estimated: 0.5 {token}\n"  
            f"Price: $2,345.67\n\n"  
            f"Confirm this trade?",  
            reply_markup=KeyboardBuilder.confirmation(f"{action}:{token}:"  
            f"{percentage}"),  
            parse_mode="HTML"  
        )
```

```
    async def handle_confirm(self, query, parts, context):  
        action_data = ":".join(parts)
```

```

await query.message.edit_text(
    "Processing trade...\n"
    "Please wait."
)

```

```

await asyncio.sleep(1)

```

```

await query.message.edit_text(
    "Trade Executed!\n\n"
    f'Action: {parts[0].upper()}\n'
    f'Token: {parts[1]}\n'
    f'Status: Complete\n'
    f'TX: 0xabc...def',
    reply_markup = InlineKeyboardMarkup([
        [InlineKeyboardButton("Back to Menu",
            callback_data="menu:main")]
    ])
)

```

```

async def handle_cancel(self, query, parts, context):
    context.user_data.pop("pending_trade", None)
    context.user_data.pop("awaiting_amount", None)

```

```

await query.message.edit_text(
    "Cancelled.",
    reply_markup = InlineKeyboardMarkup([
        [InlineKeyboardButton("Back to Menu",
            callback_data="menu:main")]
    ])
)

```

SECTION 5: CONVERSATION HANDLERS

Multi-step interactions need conversation handlers. These track user state across multiple messages, enabling flows like wallet setup, trade execution, or account configuration.

```

from telegram.ext import ConversationHandler

```

```
AWAITING_TOKEN = 1
AWAITING_AMOUNT = 2
AWAITING_ADDRESS = 3
AWAITING_CONFIRMATION = 4
```

```
class WithdrawalConversation:
```

```
    def __init__(self, bot_instance):
        self.bot = bot_instance
```

```
    def get_handler(self):
        return ConversationHandler(
            entry_points = [
                CommandHandler("withdraw", self.start_withdrawal)
            ],
            states = {
                AWAITING_TOKEN: [
                    CallbackQueryHandler(
                        self.select_token,
                        pattern = "^withdraw_token:"
                    )
                ],
                AWAITING_AMOUNT: [
                    MessageHandler(
                        filters.TEXT & ~filters.COMMAND,
                        self.enter_amount
                    )
                ],
                AWAITING_ADDRESS: [
                    MessageHandler(
                        filters.TEXT & ~filters.COMMAND,
                        self.enter_address
                    )
                ],
                AWAITING_CONFIRMATION: [
                    CallbackQueryHandler(
                        self.handle_confirmation,
                        pattern = "^withdraw_confirm:"
                    )
                ]
            ]
        )
```

```

},
fallbacks = [
    CommandHandler("cancel", self.cancel)
],
conversation_timeout = 300
)

```

```

async def start_withdrawal(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
    user_id = update.effective_user.id

```

```

    balances = await self.get_user_balances(user_id)

```

```

    if not balances:
        await update.message.reply_text(
            "You have no tokens to withdraw."
        )
    return ConversationHandler.END

```

```

keyboard = []

```

```

for token, balance in balances.items():
    keyboard.append([
        InlineKeyboardButton(
            f'{token}: {balance}',
            callback_data = f'withdraw_token:{token}'
        )
    ])

```

```

keyboard.append([
    InlineKeyboardButton("Cancel",
        callback_data = "withdraw_token:cancel")
])

```

```

await update.message.reply_text(
    "<b>Withdraw Funds</b>\n\n"
    "Select token to withdraw:",
    reply_markup = InlineKeyboardMarkup(keyboard),
    parse_mode = "HTML"
)

```

```
return AWAITING_TOKEN
```

```
async def select_token(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
    query = update.callback_query
    await query.answer()
```

```
data = query.data.split(":")[1]
```

```
if data == "cancel":
    await query.message.edit_text("Withdrawal cancelled.")
    return ConversationHandler.END
```

```
context.user_data["withdraw_token"] = data
```

```
balance = await self.get_token_balance(
    update.effective_user.id,
    data
)
```

```
await query.message.edit_text(
    f'<b>Withdraw {data}</b>\n\n'
    f'Available: {balance} {data}\n\n'
    f'Enter amount to withdraw:',
    parse_mode="HTML"
)
```

```
return AWAITING_AMOUNT
```

```
async def enter_amount(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
    text = update.message.text.strip()
```

```
try:
    amount = float(text)
except ValueError:
    await update.message.reply_text(
        "Invalid amount. Please enter a number."
    )
```

```
return AWAITING_AMOUNT
```

```
if amount <= 0:  
    await update.message.reply_text(  
        "Amount must be greater than zero."  
    )  
return AWAITING_AMOUNT
```

```
token = context.user_data["withdraw_token"]  
balance = await self.get_token_balance(  
    update.effective_user.id,  
    token  
)
```

```
if amount > balance:  
    await update.message.reply_text(  
        f'Insufficient balance. You have {balance} {token}.'  
    )  
return AWAITING_AMOUNT
```

```
context.user_data["withdraw_amount"] = amount
```

```
await update.message.reply_text(  
    f'<b>Withdraw {amount} {token}</b>\n\n'  
    f'Enter destination address:',  
    parse_mode="HTML"  
)
```

```
return AWAITING_ADDRESS
```

```
async def enter_address(self, update: Update, context:  
    ContextTypes.DEFAULT_TYPE):  
    address = update.message.text.strip()
```

```
token = context.user_data["withdraw_token"]
```

```
if not self.validate_address(address, token):  
    await update.message.reply_text(  
        "Invalid address format. Please check and try again."  
    )
```



```
return AWAITING_ADDRESS
```

```
context.user_data["withdraw_address"] = address
```

```
amount = context.user_data["withdraw_amount"]
```

```
fee = self.calculate_fee(token, amount)
```

```
short_address = f'{address[:10]}...{address[-8:]}'
```

```
await update.message.reply_text(
```

```
f'<b> Confirm Withdrawal </b> \n\n'
```

```
f'Token: {token}\n'
```

```
f'Amount: {amount}\n'
```

```
f'Fee: {fee} {token}\n'
```

```
f'You receive: {amount - fee} {token}\n'
```

```
f'To: {short_address}\n\n'
```

```
f'Confirm this withdrawal?',
```

```
reply_markup = InlineKeyboardMarkup([
```

```
[
```

```
InlineKeyboardButton(
```

```
"Confirm",
```

```
callback_data = "withdraw_confirm:yes"
```

```
),
```

```
InlineKeyboardButton(
```

```
"Cancel",
```

```
callback_data = "withdraw_confirm:no"
```

```
)
```

```
]
```

```
]),
```

```
parse_mode = "HTML"
```

```
)
```

```
return AWAITING_CONFIRMATION
```

```
async def handle_confirmation(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
```

```
query = update.callback_query
```

```
await query.answer()
```

```
decision = query.data.split(":")[1]
```

```
if decision == "no":
    await query.message.edit_text("Withdrawal cancelled.")
    self.clear_user_data(context)
    return ConversationHandler.END
```

```
token = context.user_data["withdraw_token"]
amount = context.user_data["withdraw_amount"]
address = context.user_data["withdraw_address"]
```

```
await query.message.edit_text(
    "Processing withdrawal...\n"
    "Please wait."
)
```

```
try:
    tx_hash = await self.execute_withdrawal(
        update.effective_user.id,
        token,
        amount,
        address
    )
```

```
await query.message.edit_text(
    f"<b>Withdrawal Successful!</b> \n\n"
    f"Token: {token}\n"
    f"Amount: {amount}\n"
    f"TX: <code>{tx_hash}</code> \n\n"
    f"Funds will arrive shortly.",
    parse_mode="HTML"
)
```

```
except Exception as e:
    await query.message.edit_text(
        f"<b>Withdrawal Failed</b> \n\n"
        f"Error: {str(e)}\n\n"
        f"Please try again or contact support.",
        parse_mode="HTML"
    )
```

```
self.clear_user_data(context)
return ConversationHandler.END
```

```
async def cancel(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
self.clear_user_data(context)
await update.message.reply_text("Withdrawal cancelled.")
return ConversationHandler.END
```

```
def clear_user_data(self, context):
keys = ["withdraw_token", "withdraw_amount",
"withdraw_address"]
for key in keys:
context.user_data.pop(key, None)
```

```
async def get_user_balances(self, user_id):
return {"ETH": 1.5, "USDT": 1000.0, "TON": 500.0}
```

```
async def get_token_balance(self, user_id, token):
balances = await self.get_user_balances(user_id)
return balances.get(token, 0)
```

```
def validate_address(self, address, token):
if token in ["ETH", "USDT"]:
return address.startswith("0x") and len(address) == 42
elif token == "TON":
return len(address) == 48 and address.startswith("EQ")
return False
```

```
def calculate_fee(self, token, amount):
fee_rates = {"ETH": 0.001, "USDT": 1.0, "TON": 0.05}
return fee_rates.get(token, 0.01)
```

```
async def execute_withdrawal(self, user_id, token, amount,
address):
await asyncio.sleep(2)
return "0x" + "a" * 64
```

SECTION 6: WEBHOOK DEPLOYMENT

Production bots should use webhooks instead of polling. Webhooks are more efficient, lower latency, and work with serverless deployments.

FastAPI Webhook Server:

```
from fastapi import FastAPI, Request, HTTPException
from telegram import Update, Bot
from telegram.ext import Application
import uvicorn
import os

app = FastAPI(title="Crypto Bot Webhook")

class WebhookBot:

    def __init__(self):
        self.token = os.getenv("BOT_TOKEN")
        self.webhook_url = os.getenv("WEBHOOK_URL")
        self.webhook_secret = os.getenv("WEBHOOK_SECRET", "")

        self.bot = Bot(self.token)
        self.application = None

    async def setup(self):
        self.application = (
            Application.builder()
            .token(self.token)
            .build()
        )

        from handlers.commands import setup_command_handlers
        from handlers.callbacks import setup_callback_handlers
        from handlers.conversations import setup_conversation_handlers

        setup_command_handlers(self.application)
        setup_callback_handlers(self.application)
        setup_conversation_handlers(self.application)
```

```
await self.application.initialize()
```

```
webhook_info = await self.bot.get_webhook_info()
```

```
expected_url = f"{self.webhook_url}/webhook"
```

```
if webhook_info.url != expected_url:
```

```
    await self.bot.set_webhook(
```

```
        url=expected_url,
```

```
        secret_token=self.webhook_secret,
```

```
        allowed_updates=[
```

```
            "message",
```

```
            "callback_query",
```

```
            "pre_checkout_query",
```

```
            "successful_payment"
```

```
        ],
```

```
        drop_pending_updates=True,
```

```
        max_connections=100
```

```
    )
```

```
async def shutdown(self):
```

```
    if self.application:
```

```
        await self.application.shutdown()
```

```
async def process_update(self, update_data: dict):
```

```
    update = Update.de_json(update_data, self.bot)
```

```
    await self.application.process_update(update)
```

```
webhook_bot = WebhookBot()
```

```
@app.on_event("startup")
```

```
async def startup():
```

```
    await webhook_bot.setup()
```

```
@app.on_event("shutdown")
```

```
async def shutdown():
```

```
    await webhook_bot.shutdown()
```

```
@app.post("/webhook")
```

```
async def webhook_handler(request: Request):
```

```
secret_token = request.headers.get("X-Telegram-Bot-API-Secret-Token", "")
```

```
if webhook_bot.webhook_secret and secret_token !=  
webhook_bot.webhook_secret:  
    raise HTTPException(status_code=401, detail="Invalid secret  
token")
```

```
try:  
    update_data = await request.json()  
    await webhook_bot.process_update(update_data)  
    return {"status": "ok"}  
except Exception as e:  
    print(f"Webhook error: {e}")  
    raise HTTPException(status_code=500, detail=str(e))
```

```
@app.get("/health")  
async def health():  
    return {"status": "healthy"}
```

```
@app.get("/webhook/info")  
async def webhook_info():  
    info = await webhook_bot.bot.get_webhook_info()  
    return {  
        "url": info.url,  
        "pending_updates": info.pending_update_count,  
        "last_error": info.last_error_message,  
        "max_connections": info.max_connections  
    }
```

```
if __name__ == "__main__":  
    uvicorn.run(  
        "main:app",  
        host="0.0.0.0",  
        port=int(os.getenv("PORT", 8000)),  
        reload=os.getenv("ENVIRONMENT") == "development"  
    )
```

Nginx Configuration for Webhooks:

```

server {
    listen 443 ssl http2;
    server_name bot.yourdomain.com;

    ssl_certificate /etc/letsencrypt/live/bot.yourdomain.com/
fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/bot.yourdomain.com/
privkey.pem;

    ssl_protocols TLSv1.2 TLSv1.3;
    ssl_ciphers ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-
AES128-GCM-SHA256;
    ssl_prefer_server_ciphers off;

    location /webhook {
        proxy_pass http://127.0.0.1:8000;
        proxy_http_version 1.1;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;

        proxy_connect_timeout 5s;
        proxy_send_timeout 10s;
        proxy_read_timeout 30s;
    }

    location /health {
        proxy_pass http://127.0.0.1:8000;
        proxy_http_version 1.1;
    }
}

```

SECTION 7: ERROR HANDLING AND RESILIENCE

Production bots need robust error handling. Users shouldn't see stack traces. Failures should be logged. Recovery should be automatic.

Comprehensive Error Handler:

```
from telegram.error import (
    TelegramError,
    Forbidden,
    NetworkError,
    BadRequest,
    TimedOut,
    RetryAfter,
    InvalidToken
)
import traceback
import structlog

logger = structlog.get_logger()

class ErrorHandler:

    def __init__(self, admin_user_ids: list = None):
        self.admin_user_ids = admin_user_ids or []
        self.error_counts = {}
        self.error_threshold = 10
        self.error_window = 300

    async def handle_error(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
        error = context.error

        log_data = {
            "error_type": type(error).__name__,
            "error_message": str(error),
            "update_id": update.update_id if update else None,
            "user_id": update.effective_user.id if update and
update.effective_user else None,
            "chat_id": update.effective_chat.id if update and
update.effective_chat else None
        }

        if isinstance(error, Forbidden):
```



```
logger.warning("user_blocked_bot", **log_data)
return
```

```
if isinstance(error, InvalidToken):
logger.critical("invalid_bot_token", **log_data)
return
```

```
if isinstance(error, RetryAfter):
logger.warning(
"rate_limited",
retry_after = error.retry_after,
**log_data
)
await asyncio.sleep(error.retry_after)
return
```

```
if isinstance(error, Timeout):
logger.warning("request_timed_out", **log_data)
return
```

```
if isinstance(error, NetworkError):
logger.error("network_error", **log_data)
await self.notify_user_of_error(update)
return
```

```
if isinstance(error, BadRequest):
logger.error("bad_request", **log_data)
```

```
if "message is not modified" in str(error):
return
```

```
if "query is too old" in str(error):
if update and update.callback_query:
await update.callback_query.answer(
"This button has expired. Please try again."
)
return
```

```
await self.notify_user_of_error(update)
return
```

```
logger.error(  
    "unhandled_error",  
    traceback = traceback.format_exc(),  
    **log_data  
)
```

```
await self.notify_user_of_error(update)
```

```
if self.should_notify_admins(error):  
    await self.notify_admins(error, update, context)
```

```
async def notify_user_of_error(self, update: Update):  
    if not update:  
        return
```

```
    try:  
        if update.callback_query:  
            await update.callback_query.answer(  
                "An error occurred. Please try again.",  
                show_alert = True  
            )  
        elif update.effective_message:  
            await update.effective_message.reply_text(  
                "An error occurred. Please try again later.\n"  
                "If the problem persists, contact support."  
            )  
    except TelegramError:  
        pass
```

```
def should_notify_admins(self, error):  
    error_key = type(error).__name__  
    now = time.time()
```

```
    if error_key not in self.error_counts:  
        self.error_counts[error_key] = []
```

```
    self.error_counts[error_key] = [  
        ts for ts in self.error_counts[error_key]  
        if now - ts < self.error_window
```

```
]
```

```
self.error_counts[error_key].append(now)
```

```
if len(self.error_counts[error_key]) >= self.error_threshold:
```

```
self.error_counts[error_key] = []
```

```
return True
```

```
return False
```

```
async def notify_admins(self, error, update: Update, context:
ContextTypes.DEFAULT_TYPE):
```

```
error_text = (
```

```
f'<b> Bot Error Alert </b> \n\n"
```

```
f"Type: {type(error)._name_}\n"
```

```
f"Message: {str(error)[:200]}\n"
```

```
f"User: {update.effective_user.id if update and update.effective_user
else 'N/A'}\n"
```

```
f"Time: {time.strftime('%Y-%m-%d %H:%M:%S UTC',
time.gmtime())}"
```

```
)
```

```
for admin_id in self.admin_user_ids:
```

```
try:
```

```
await context.bot.send_message(
```

```
chat_id = admin_id,
```

```
text = error_text,
```

```
parse_mode = "HTML"
```

```
)
```

```
except TelegramError:
```

```
pass
```

SECTION 8: RATE LIMITING AND THROTTLING

Telegram enforces rate limits. Your bot must respect them to avoid being blocked.

```
from collections import defaultdict
import asyncio
```

```
import time
```

```
class TelegramRateLimiter:
```

```
    def __init__(self):
```

```
        self.global_limit = 30
```

```
        self.global_tokens = self.global_limit
```

```
        self.global_last_update = time.time()
```

```
        self.chat_limits = defaultdict(lambda: {
```

```
            "tokens": 20,
```

```
            "last_update": time.time()
```

```
        })
```

```
        self.user_limits = defaultdict(lambda: {
```

```
            "tokens": 1,
```

```
            "last_update": time.time()
```

```
        })
```

```
        self.chat_rate = 20
```

```
        self.chat_window = 60
```

```
        self.user_rate = 1
```

```
        self.user_window = 1
```

```
        self.lock = asyncio.Lock()
```

```
    async def acquire(self, chat_id: int, is_group: bool = False):
```

```
        async with self.lock:
```

```
            await self._refill_global()
```

```
            await self._wait_for_global()
```

```
        if is_group:
```

```
            await self._refill_chat(chat_id)
```

```
            await self._wait_for_chat(chat_id)
```

```
        else:
```

```
            await self._refill_user(chat_id)
```

```
            await self._wait_for_user(chat_id)
```

```
    async def _refill_global(self):
```

```
now = time.time()
elapsed = now - self.global_last_update
self.global_tokens = min(
self.global_limit,
self.global_tokens + elapsed * self.global_limit
)
self.global_last_update = now
```

```
async def _wait_for_global(self):
if self.global_tokens < 1:
wait_time = (1 - self.global_tokens) / self.global_limit
await asyncio.sleep(wait_time)
self.global_tokens = 0
else:
self.global_tokens -= 1
```

```
async def _refill_chat(self, chat_id: int):
now = time.time()
chat_data = self.chat_limits[chat_id]
elapsed = now - chat_data["last_update"]
chat_data["tokens"] = min(
self.chat_rate,
chat_data["tokens"] + elapsed * (self.chat_rate / self.chat_window)
)
chat_data["last_update"] = now
```

```
async def _wait_for_chat(self, chat_id: int):
chat_data = self.chat_limits[chat_id]
```

```
if chat_data["tokens"] < 1:
wait_time = (1 - chat_data["tokens"]) * (self.chat_window /
self.chat_rate)
await asyncio.sleep(wait_time)
chat_data["tokens"] = 0
else:
chat_data["tokens"] -= 1
```

```
async def _refill_user(self, user_id: int):
now = time.time()
user_data = self.user_limits[user_id]
```

```
elapsed = now - user_data["last_update"]
user_data["tokens"] = min(
    self.user_rate,
    user_data["tokens"] + elapsed * self.user_rate
)
user_data["last_update"] = now
```

```
async def _wait_for_user(self, user_id: int):
    user_data = self.user_limits[user_id]
```

```
if user_data["tokens"] < 1:
    wait_time = (1 - user_data["tokens"]) / self.user_rate
    await asyncio.sleep(wait_time)
    user_data["tokens"] = 0
else:
    user_data["tokens"] -= 1
```

```
class ThrottledMessageSender:
```

```
    def __init__(self, bot: Bot):
        self.bot = bot
        self.rate_limiter = TelegramRateLimiter()
```

```
    async def send_message(
        self,
        chat_id: int,
        text: str,
        is_group: bool = False,
        **kwargs
    ):
        await self.rate_limiter.acquire(chat_id, is_group)
```

```
        return await self.bot.send_message(
            chat_id=chat_id,
            text=text,
            **kwargs
        )
```

```
    async def broadcast(
```

```

self,
chat_ids: list,
text: str,
delay: float = 0.05
):
    results = {"success": 0, "failed": 0, "blocked": 0}

    for chat_id in chat_ids:
        try:
            await self.send_message(chat_id, text)
            results["success"] += 1
        except Forbidden:
            results["blocked"] += 1
        except TelegramError:
            results["failed"] += 1

    await asyncio.sleep(delay)

    return results

```

SECTION 9: DATABASE INTEGRATION

Bots need persistent storage. PostgreSQL handles user data, settings, and history. Redis handles sessions and caching.

PostgreSQL Models:

```

import asyncpg
from datetime import datetime
from typing import Optional, List, Dict, Any

class Database:

    def __init__(self, database_url: str):
        self.database_url = database_url
        self.pool = None

    async def connect(self):
        self.pool = await asyncpg.create_pool(

```

```
self.database_url,  
min_size = 5,  
max_size = 20  
)
```

```
async def disconnect(self):  
if self.pool:  
await self.pool.close()
```

```
async def execute(self, query: str, *args):  
async with self.pool.acquire() as conn:  
return await conn.execute(query, *args)
```

```
async def fetch(self, query: str, *args):  
async with self.pool.acquire() as conn:  
return await conn.fetch(query, *args)
```

```
async def fetchrow(self, query: str, *args):  
async with self.pool.acquire() as conn:  
return await conn.fetchrow(query, *args)
```

```
async def fetchval(self, query: str, *args):  
async with self.pool.acquire() as conn:  
return await conn.fetchval(query, *args)
```

```
class UserRepository:
```

```
def __init__(self, db: Database):  
self.db = db
```

```
async def create_tables(self):  
await self.db.execute("""  
CREATE TABLE IF NOT EXISTS users (  
user_id BIGINT PRIMARY KEY,  
username VARCHAR(255),  
first_name VARCHAR(255),  
language_code VARCHAR(10),  
created_at TIMESTAMP DEFAULT NOW(),  
updated_at TIMESTAMP DEFAULT NOW(),
```



```
is_active BOOLEAN DEFAULT TRUE,  
settings JSONB DEFAULT '{}':jsonb  
)  
""")
```

```
await self.db.execute("""  
CREATE TABLE IF NOT EXISTS user_wallets (  
id SERIAL PRIMARY KEY,  
user_id BIGINT REFERENCES users(user_id),  
chain VARCHAR(50),  
address VARCHAR(100),  
is_primary BOOLEAN DEFAULT FALSE,  
created_at TIMESTAMP DEFAULT NOW(),  
UNIQUE(user_id, chain, address)  
)  
""")
```

```
await self.db.execute("""  
CREATE TABLE IF NOT EXISTS transactions (  
id SERIAL PRIMARY KEY,  
user_id BIGINT REFERENCES users(user_id),  
type VARCHAR(50),  
token VARCHAR(20),  
amount DECIMAL(36, 18),  
tx_hash VARCHAR(100),  
status VARCHAR(20),  
created_at TIMESTAMP DEFAULT NOW(),  
completed_at TIMESTAMP  
)  
""")
```

```
await self.db.execute("""  
CREATE INDEX IF NOT EXISTS idx_transactions_user_id  
ON transactions(user_id)  
""")
```

```
await self.db.execute("""  
CREATE INDEX IF NOT EXISTS idx_transactions_created_at  
ON transactions(created_at DESC)  
""")
```

```

async def get_user(self, user_id: int) -> Optional[Dict[str, Any]]:
    row = await self.db.fetchrow(
        "SELECT * FROM users WHERE user_id = $1",
        user_id
    )
    return dict(row) if row else None

```

```

async def create_user(
    self,
    user_id: int,
    username: str = None,
    first_name: str = None,
    language_code: str = None
) -> Dict[str, Any]:
    row = await self.db.fetchrow("""
INSERT INTO users (user_id, username, first_name, language_code)
VALUES ($1, $2, $3, $4)
ON CONFLICT (user_id) DO UPDATE SET
username = EXCLUDED.username,
first_name = EXCLUDED.first_name,
updated_at = NOW()
RETURNING *
""", user_id, username, first_name, language_code)

    return dict(row)

```

```

async def update_settings(
    self,
    user_id: int,
    settings: Dict[str, Any]
):
    await self.db.execute("""
UPDATE users
SET settings = settings || $2::jsonb, updated_at = NOW()
WHERE user_id = $1
""", user_id, settings)

```

```

async def get_user_wallets(self, user_id: int) -> List[Dict[str, Any]]:
    rows = await self.db.fetch("""

```

```
SELECT * FROM user_wallets
WHERE user_id = $1
ORDER BY is_primary DESC, created_at ASC
""", user_id)
```

```
return [dict(row) for row in rows]
```

```
async def add_wallet(
    self,
    user_id: int,
    chain: str,
    address: str,
    is_primary: bool = False
):
    if is_primary:
        await self.db.execute("""
UPDATE user_wallets
SET is_primary = FALSE
WHERE user_id = $1 AND chain = $2
""", user_id, chain)
```

```
await self.db.execute("""
INSERT INTO user_wallets (user_id, chain, address, is_primary)
VALUES ($1, $2, $3, $4)
ON CONFLICT (user_id, chain, address) DO UPDATE SET
is_primary = EXCLUDED.is_primary
""", user_id, chain, address, is_primary)
```

```
async def record_transaction(
    self,
    user_id: int,
    tx_type: str,
    token: str,
    amount: float,
    tx_hash: str = None,
    status: str = "pending"
) -> int:
    tx_id = await self.db.fetchval("""
INSERT INTO transactions
(user_id, type, token, amount, tx_hash, status)
```

```
VALUES ($1, $2, $3, $4, $5, $6)
RETURNING id
""", user_id, tx_type, token, amount, tx_hash, status)
```

```
return tx_id
```

```
async def update_transaction_status(
    self,
    tx_id: int,
    status: str,
    tx_hash: str = None
):
    await self.db.execute("""
UPDATE transactions SET
status = $2,
tx_hash = COALESCE($3, tx_hash),
completed_at = CASE WHEN $2 IN ('completed', 'failed') THEN
NOW() ELSE completed_at END
WHERE id = $1
""", tx_id, status, tx_hash)
```

SECTION 10: TESTING BOT HANDLERS

Testing ensures reliability. Mock the Telegram API to test handler logic.

```
import pytest
from unittest.mock import AsyncMock, MagicMock, patch
from telegram import Update, User, Chat, Message
```

```
class TestFactory:
```

```
@staticmethod
def create_user(
    user_id: int = 12345,
    username: str = "testuser",
    first_name: str = "Test"
) -> User:
    return User(
```

```
id = user_id,  
first_name = first_name,  
is_bot = False,  
username = username  
)
```

```
@staticmethod  
def create_chat(  
chat_id: int = 12345,  
chat_type: str = "private"  
) -> Chat:  
return Chat(id = chat_id, type = chat_type)
```

```
@staticmethod  
def create_message(  
text: str,  
user_id: int = 12345,  
chat_id: int = 12345  
) -> MagicMock:  
user = TestFactory.create_user(user_id)  
chat = TestFactory.create_chat(chat_id)
```

```
message = MagicMock(spec = Message)  
message.text = text  
message.from_user = user  
message.chat = chat  
message.reply_text = AsyncMock()  
message.edit_text = AsyncMock()  
message.delete = AsyncMock()
```

```
return message
```

```
@staticmethod  
def create_update(  
text: str,  
user_id: int = 12345,  
chat_id: int = 12345  
) -> MagicMock:  
message = TestFactory.create_message(text, user_id, chat_id)
```

```
update = MagicMock(spec=Update)
update.message = message
update.effective_user = message.from_user
update.effective_chat = message.chat
update.effective_message = message
update.callback_query = None
```

```
return update
```

```
@staticmethod
def create_context(
    args: list = None,
    user_data: dict = None
) -> MagicMock:
    context = MagicMock()
    context.args = args or []
    context.user_data = user_data or {}
    context.chat_data = {}
    context.bot = AsyncMock()
```

```
return context
```

```
class TestPriceCommand:
```

```
@pytest.fixture
def bot(self):
    return CryptoBot()
```

```
@pytest.mark.asyncio
async def test_price_command_with_valid_symbol(self, bot):
    update = TestFactory.create_update("/price ETH")
    context = TestFactory.create_context(args=["ETH"])
```

```
await bot.cmd_price(update, context)
```

```
update.message.reply_text.assert_called_once()
```

```
call_text = update.message.reply_text.call_args[0][0]
assert "ETH" in call_text
```

```
assert "$" in call_text
```

```
@pytest.mark.asyncio
async def test_price_command_without_symbol(self, bot):
    update = TestFactory.create_update("/price")
    context = TestFactory.create_context(args = [])
```

```
await bot.cmd_price(update, context)
```

```
call_text = update.message.reply_text.call_args[0][0]
assert "Usage" in call_text
```

```
@pytest.mark.asyncio
async def test_price_command_invalid_symbol(self, bot):
    update = TestFactory.create_update("/price INVALID")
    context = TestFactory.create_context(args = ["INVALID"])
```

```
await bot.cmd_price(update, context)
```

```
call_text = update.message.reply_text.call_args[0][0]
assert "Could not find" in call_text
```

```
class TestCallbackHandler:
```

```
@pytest.fixture
def handler(self):
    return CallbackHandler(MagicMock())
```

```
@pytest.mark.asyncio
async def test_menu_callback(self, handler):
    query = MagicMock()
    query.data = "menu:wallet"
    query.answer = AsyncMock()
    query.message.edit_text = AsyncMock()
```

```
await handler.handle_menu(query, ["wallet"], MagicMock())
```

```
query.message.edit_text.assert_called_once()
```

```
call_kwargs = query.message.edit_text.call_args
assert "Wallet" in call_kwargs[0][0]
```

Running tests:

```
pytest tests/ -v --asyncio-mode=auto
```

CHAPTER SUMMARY

This chapter covered bot development fundamentals:

BotFather creates and configures bots. Tokens authenticate your code. Commands populate the menu.

python-telegram-bot v22 uses Application-centered architecture. Handlers route updates. Context persists data.

aiogram offers middleware-based design. Routers organize handlers. Filters target specific updates.

Inline keyboards create interactive interfaces. Callback queries handle button clicks. Edit messages to update content.

Conversation handlers manage multi-step flows. States track progress. Timeouts prevent stuck conversations.

Webhooks beat polling for production. HTTPS required. Secret tokens verify authenticity.

Error handling prevents crashes. Log everything. Notify users gracefully. Alert admins on critical failures.

Rate limiting respects Telegram's constraints. Token buckets smooth traffic. Delays prevent blocks.

Database integration persists data. PostgreSQL handles structured data. Redis handles sessions and cache.

Testing ensures reliability. Mock the API. Test handler logic. Verify error paths.

Next chapter: Mini Apps Development. We build full-screen web applications within Telegram.

CHAPTER 3: MINI APPS DEVELOPMENT

Building full-screen applications within Telegram. Mini Apps 2.0 enables games, wallets, trading interfaces, and complex applications. This chapter covers the Web Apps API, device sensors, storage systems, and TON Connect integration.

SECTION 1: MINI APPS ARCHITECTURE

Mini Apps are web applications running inside Telegram. They're not native apps - they're standard HTML, CSS, and JavaScript rendered in a WebView. But they access Telegram features through dedicated APIs.

The execution environment:

User taps Mini App link or button in bot. Telegram opens WebView container. Your web app loads from your server. The app calls `Telegram.WebApp.ready()`. Telegram injects initialization data. Your app can now access Telegram features.

What Mini Apps can do:

- Access user information (with permission)
- Send data back to the bot
- Process payments with Telegram Stars
- Connect to TON wallets
- Use device sensors (accelerometer, gyroscope)
- Store data locally (DeviceStorage, SecureStorage)
- Enter full-screen mode
- Lock screen orientation
- Share content to chats and stories

What Mini Apps cannot do:

- Access device camera or microphone directly
- Send push notifications independently
- Run in background when closed
- Access other apps or system features
- Bypass Telegram's security sandbox

Project Structure:

```
telegram-mini-app/  
  public/  
    index.html  
    manifest.json  
    tonconnect-manifest.json  
  src/  
    main.js  
    app.js  
    components/  
      wallet.js  
      trading.js  
      settings.js  
    services/  
      telegram.js  
      ton.js  
      api.js  
    styles/  
      main.css  
      theme.css  
  package.json  
  vite.config.js
```

Basic HTML Structure:

The index.html file loads Telegram's Web Apps script:

```
<!DOCTYPE html >  
<html lang="en">  
<head>  
  <meta charset="UTF-8">
```

```
<meta name="viewport" content="width=device-width, initial-
scale=1.0, maximum-scale=1.0, user-scalable=no">
<title>Crypto Mini App</title>
<script src="https://telegram.org/js/telegram-web-app.js"></
script>
<link rel="stylesheet" href="/src/styles/main.css">
</head>
<body>
<div id="app">
<div class="loading">Loading...</div>
</div>
<script type="module" src="/src/main.js"></script>
</body>
</html>
```

CSS Theme Variables:

Mini Apps should respect Telegram's theme. Use CSS variables that update automatically:

```
:root {
--tg-theme-bg-color: #ffffff;
--tg-theme-text-color: #000000;
--tg-theme-hint-color: #999999;
--tg-theme-link-color: #2481cc;
--tg-theme-button-color: #2481cc;
--tg-theme-button-text-color: #ffffff;
--tg-theme-secondary-bg-color: #f4f4f4;

--safe-area-top: 0px;
--safe-area-bottom: 0px;
--safe-area-left: 0px;
--safe-area-right: 0px;
}

* {
box-sizing: border-box;
margin: 0;
padding: 0;
```

```
}
```

```
body {  
  font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI',  
  Roboto, sans-serif;  
  background-color: var(--tg-theme-bg-color);  
  color: var(--tg-theme-text-color);  
  min-height: 100vh;  
  overflow-x: hidden;  
}
```

```
#app {  
  padding: var(--safe-area-top) var(--safe-area-right) var(--safe-area-  
bottom) var(--safe-area-left);  
  min-height: 100vh;  
}
```

```
.button {  
  background-color: var(--tg-theme-button-color);  
  color: var(--tg-theme-button-text-color);  
  border: none;  
  border-radius: 8px;  
  padding: 12px 24px;  
  font-size: 16px;  
  font-weight: 600;  
  cursor: pointer;  
  transition: opacity 0.2s;  
}
```

```
.button:active {  
  opacity: 0.8;  
}
```

```
.button:disabled {  
  opacity: 0.5;  
  cursor: not-allowed;  
}
```

```
.card {  
  background-color: var(--tg-theme-secondary-bg-color);
```

```

border-radius: 12px;
padding: 16px;
margin: 8px 0;
}

.hint {
  color: var(--tg-theme-hint-color);
  font-size: 14px;
}

.link {
  color: var(--tg-theme-link-color);
  text-decoration: none;
}

```

SECTION 2: TELEGRAM WEB APP API

The `window.Telegram.WebApp` object exposes all Mini App functionality. Understanding its properties and methods is essential.

Core Telegram Service:

```

class TelegramService {

  constructor() {
    this.tg = window.Telegram.WebApp;
    this.user = null;
    this.initData = null;
    this.isReady = false;
  }

  init() {
    if (!this.tg) {
      console.error('Telegram WebApp not available');
      return false;
    }

    this.tg.ready();
    this.tg.expand();
  }
}

```

```
this.initData = this.tg.initDataUnsafe;  
this.user = this.initData.user;
```

```
this.applyTheme();
```

```
this.setupEventListeners();
```

```
this.isReady = true;
```

```
return true;  
}
```

```
applyTheme() {  
  const theme = this.tg.themeParams;  
  const root = document.documentElement;
```

```
  root.style.setProperty('--tg-theme-bg-color', theme.bg_color ||  
  '#ffffff');  
  root.style.setProperty('--tg-theme-text-color', theme.text_color ||  
  '#000000');  
  root.style.setProperty('--tg-theme-hint-color', theme.hint_color ||  
  '#999999');  
  root.style.setProperty('--tg-theme-link-color', theme.link_color ||  
  '#2481cc');  
  root.style.setProperty('--tg-theme-button-color', theme.button_color  
  || '#2481cc');  
  root.style.setProperty('--tg-theme-button-text-color',  
  theme.button_text_color || '#ffffff');  
  root.style.setProperty('--tg-theme-secondary-bg-color',  
  theme.secondary_bg_color || '#f4f4f4');  
}
```

```
setupEventListeners() {  
  this.tg.onEvent('themeChanged', () => {  
    this.applyTheme();  
    this.onThemeChanged();  
  });
```

```
this.tg.onEvent('viewportChanged', (event) => {
```

```
this.onViewportChanged(event);
});
```

```
this.tg.onEvent('mainButtonClicked', () => {
  this.onMainButtonClicked();
});
```

```
this.tg.onEvent('backButtonClicked', () => {
  this.onBackButtonClicked();
});
```

```
this.tg.onEvent('settingsButtonClicked', () => {
  this.onSettingsButtonClicked();
});
}
```

```
onThemeChanged() {
}
```

```
onViewportChanged(event) {
}
```

```
onMainButtonClicked() {
}
```

```
onBackButtonClicked() {
}
```

```
onSettingsButtonClicked() {
}
```

```
getUser() {
  return this.user;
}
```

```
getUserId() {
  return this.user?.id;
}
```

```
getUsername() {
```

```
return this.user?.username;  
}
```

```
getFirstName() {  
return this.user?.first_name;  
}
```

```
getLanguageCode() {  
return this.user?.language_code || 'en';  
}
```

```
isPremium() {  
return this.user?.is_premium || false;  
}
```

```
getPlatform() {  
return this.tg.platform;  
}
```

```
getVersion() {  
return this.tg.version;  
}
```

```
isVersionAtLeast(version) {  
return this.tg.isVersionAtLeast(version);  
}
```

```
getColorScheme() {  
return this.tg.colorScheme;  
}
```

```
getInitData() {  
return this.tg.initData;  
}
```

```
getInitDataUnsafe() {  
return this.tg.initDataUnsafe;  
}
```

```
showMainButton(text, onClick = null) {
```



```
this.tg.MainButton.setText(text);
this.tg.MainButton.show();

if (onClick) {
  this.mainButtonCallback = onClick;
}

hideMainButton() {
  this.tg.MainButton.hide();
}

setMainButtonLoading(loading) {
  if (loading) {
    this.tg.MainButton.showProgress();
  } else {
    this.tg.MainButton.hideProgress();
  }
}

enableMainButton() {
  this.tg.MainButton.enable();
}

disableMainButton() {
  this.tg.MainButton.disable();
}

showBackButton() {
  this.tg.BackButton.show();
}

hideBackButton() {
  this.tg.BackButton.hide();
}

showAlert(message) {
  return new Promise((resolve) => {
    this.tg.showAlert(message, resolve);
  });
}
```

```
}
```

```
showConfirm(message) {  
  return new Promise((resolve) => {  
    this.tg.showConfirm(message, resolve);  
  });  
}
```

```
showPopup(params) {  
  return new Promise((resolve) => {  
    this.tg.showPopup(params, (buttonId) => {  
      resolve(buttonId);  
    });  
  });  
}
```

```
hapticFeedback(type, style = null) {  
  if (!this.tg.HapticFeedback) return;
```

```
  switch (type) {  
    case 'impact':  
      this.tg.HapticFeedback.impactOccurred(style || 'medium');  
      break;  
    case 'notification':  
      this.tg.HapticFeedback.notificationOccurred(style || 'success');  
      break;  
    case 'selection':  
      this.tg.HapticFeedback.selectionChanged();  
      break;  
  }  
}
```

```
sendData(data) {  
  this.tg.sendData(JSON.stringify(data));  
}
```

```
close() {  
  this.tg.close();  
}
```

```
openLink(url, options = {}) {  
  this.tg.openLink(url, options);  
}
```

```
openTelegramLink(url) {  
  this.tg.openTelegramLink(url);  
}
```

```
openInvoice(url) {  
  return new Promise((resolve) => {  
    this.tg.openInvoice(url, (status) => {  
      resolve(status);  
    });  
  });  
}
```

```
requestWriteAccess() {  
  return new Promise((resolve) => {  
    this.tg.requestWriteAccess((granted) => {  
      resolve(granted);  
    });  
  });  
}
```

```
requestContact() {  
  return new Promise((resolve) => {  
    this.tg.requestContact((result) => {  
      resolve(result);  
    });  
  });  
}
```

```
setHeaderColor(color) {  
  this.tg.setHeaderColor(color);  
}
```

```
setBackgroundColor(color) {  
  this.tg.setBackgroundColor(color);  
}  
}
```

```
const telegramService = new TelegramService();  
export default telegramService;
```

SECTION 3: FULL-SCREEN MODE

Mini Apps 2.0 support full-screen experiences. Games, video players, and immersive interfaces can fill the entire screen.

Full-Screen Manager:

```
class FullScreenManager {  
  
  constructor(telegramService) {  
    this.tg = telegramService.tg;  
    this.isFullScreen = false;  
    this.orientationLocked = false;  
    this.onFullScreenChange = null;  
  }  
  
  isSupported() {  
    return this.tg.isVersionAtLeast('8.0');  
  }  
  
  async enterFullScreen() {  
    if (!this.isSupported()) {  
      console.warn('Full screen not supported');  
      return false;  
    }  
  
    try {  
      await this.tg.requestFullscreen();  
      this.isFullScreen = true;  
      this.applySafeAreaInsets();  
    }  
  
    if (this.onFullScreenChange) {  
      this.onFullScreenChange(true);  
    }  
  }  
}
```

```
return true;
} catch (error) {
  console.error('Failed to enter fullscreen:', error);
  return false;
}
}
```

```
async exitFullScreen() {
  if (!this.isFullScreen) return;
```

```
  try {
    await this.tg.exitFullscreen();
    this.isFullScreen = false;
    this.removeSafeAreaInsets();
```

```
  if (this.onFullScreenChange) {
    this.onFullScreenChange(false);
  }
} catch (error) {
  console.error('Failed to exit fullscreen:', error);
}
}
```

```
toggle() {
  if (this.isFullScreen) {
    return this.exitFullScreen();
  } else {
    return this.enterFullScreen();
  }
}
```

```
applySafeAreaInsets() {
  const safeArea = this.tg.safeAreaInset || { top: 0, bottom: 0, left: 0,
right: 0 };
  const contentSafeArea = this.tg.contentSafeAreaInset || { top: 0,
bottom: 0, left: 0, right: 0 };

```

```
const root = document.documentElement;
```

```
root.style.setProperty('--safe-area-top', `${safeArea.top}px`);
```

```
root.style.setProperty('--safe-area-bottom', `${safeArea.bottom}px`);
root.style.setProperty('--safe-area-left', `${safeArea.left}px`);
root.style.setProperty('--safe-area-right', `${safeArea.right}px`);
```

```
root.style.setProperty('--content-safe-top', `${contentSafeArea.top}
px`);
root.style.setProperty('--content-safe-bottom', `
${contentSafeArea.bottom}px`);
```

```
document.body.classList.add('fullscreen-mode');
}
```

```
removeSafeAreaInsets() {
const root = document.documentElement;

root.style.setProperty('--safe-area-top', '0px');
root.style.setProperty('--safe-area-bottom', '0px');
root.style.setProperty('--safe-area-left', '0px');
root.style.setProperty('--safe-area-right', '0px');
```

```
document.body.classList.remove('fullscreen-mode');
}
```

```
lockOrientation(orientation) {
if (!this.tg.isVersionAtLeast('8.0')) return;
```

```
this.tg.lockOrientation(orientation);
this.orientationLocked = true;
}
```

```
unlockOrientation() {
if (!this.orientationLocked) return;
```

```
this.tg.unlockOrientation();
this.orientationLocked = false;
}
```

```
getIsFullScreen() {
return this.isFullScreen;
}
```

```
}
```

Full-Screen CSS:

```
.fullscreen-mode {  
  padding: 0 !important;  
}
```

```
.fullscreen-mode .app-container {  
  padding-top: var(--safe-area-top);  
  padding-bottom: var(--safe-area-bottom);  
  padding-left: var(--safe-area-left);  
  padding-right: var(--safe-area-right);  
}
```

```
.fullscreen-mode .header {  
  padding-top: calc(var(--safe-area-top) + 16px);  
}
```

```
.fullscreen-mode .bottom-nav {  
  padding-bottom: calc(var(--safe-area-bottom) + 16px);  
}
```

```
.fullscreen-mode .game-canvas {  
  width: 100vw;  
  height: 100vh;  
  position: fixed;  
  top: 0;  
  left: 0;  
}
```

```
.fullscreen-controls {  
  position: fixed;  
  top: var(--safe-area-top);  
  right: var(--safe-area-right);  
  padding: 16px;  
  z-index: 1000;  
}
```

```
.exit-fullscreen-btn {
background: rgba(0, 0, 0, 0.5);
color: white;
border: none;
border-radius: 50%;
width: 40px;
height: 40px;
font-size: 20px;
cursor: pointer;
}
```

SECTION 4: DEVICE SENSORS

Mini Apps can access accelerometer, gyroscope, and device orientation. This enables motion-controlled games and immersive experiences.

Sensor Manager:

```
class SensorManager {

constructor(telegramService) {
this.tg = telegramService.tg;
this.accelerometer = null;
this.gyroscope = null;
this.deviceOrientation = null;

this.accelerometerData = { x: 0, y: 0, z: 0 };
this.gyroscopeData = { x: 0, y: 0, z: 0 };
this.orientationData = { alpha: 0, beta: 0, gamma: 0 };

this.listeners = {
accelerometer: [],
gyroscope: [],
orientation: []
};
}

isAccelerometerAvailable() {
```



```
return !!this.tg.Accelerometer;  
}
```

```
isGyroscopeAvailable() {  
return !!this.tg.Gyroscope;  
}
```

```
isDeviceOrientationAvailable() {  
return !!this.tg.DeviceOrientation;  
}
```

```
startAccelerometer(refreshRate = 100) {  
if (!this.isAccelerometerAvailable()) {  
console.warn('Accelerometer not available');  
return false;  
}
```

```
this.accelerometer = this.tg.Accelerometer;
```

```
this.accelerometer.start({ refresh_rate: refreshRate });
```

```
this.accelerometer.onData = (data) => {  
this.accelerometerData = {  
x: data.x,  
y: data.y,  
z: data.z  
};
```

```
this.notifyListeners('accelerometer', this.accelerometerData);  
};
```

```
return true;  
}
```

```
stopAccelerometer() {  
if (this.accelerometer) {  
this.accelerometer.stop();  
this.accelerometer = null;  
}  
}
```

```
startGyroscope(refreshRate = 100) {  
  if (!this.isGyroscopeAvailable()) {  
    console.warn('Gyroscope not available');  
    return false;  
  }  
}
```

```
this.gyroscope = this.tg.Gyroscope;
```

```
this.gyroscope.start({ refresh_rate: refreshRate });
```

```
this.gyroscope.onData = (data) => {  
  this.gyroscopeData = {  
    x: data.x,  
    y: data.y,  
    z: data.z  
  };  
};
```

```
this.notifyListeners('gyroscope', this.gyroscopeData);  
};
```

```
return true;  
}
```

```
stopGyroscope() {  
  if (this.gyroscope) {  
    this.gyroscope.stop();  
    this.gyroscope = null;  
  }  
}
```

```
startDeviceOrientation(refreshRate = 100, absolute = false) {  
  if (!this.isDeviceOrientationAvailable()) {  
    console.warn('Device orientation not available');  
    return false;  
  }  
}
```

```
this.deviceOrientation = this.tg.DeviceOrientation;
```

```
this.deviceOrientation.start({
```

```
refresh_rate: refreshRate,  
need_absolute: absolute  
});
```

```
this.deviceOrientation.onData = (data) => {  
this.orientationData = {  
alpha: data.alpha,  
beta: data.beta,  
gamma: data.gamma,  
absolute: data.absolute  
};
```

```
this.notifyListeners('orientation', this.orientationData);  
};
```

```
return true;  
}
```

```
stopDeviceOrientation() {  
if (this.deviceOrientation) {  
this.deviceOrientation.stop();  
this.deviceOrientation = null;  
}  
}
```

```
startAll(refreshRate = 100) {  
this.startAccelerometer(refreshRate);  
this.startGyroscope(refreshRate);  
this.startDeviceOrientation(refreshRate);  
}
```

```
stopAll() {  
this.stopAccelerometer();  
this.stopGyroscope();  
this.stopDeviceOrientation();  
}
```

```
onAccelerometer(callback) {  
this.listeners.accelerometer.push(callback);  
return () => {
```

```
this.listeners.accelerometer = this.listeners.accelerometer.filter(
  cb => cb !== callback
);
};
}
```

```
onGyroscope(callback) {
  this.listeners.gyroscope.push(callback);
  return () => {
    this.listeners.gyroscope = this.listeners.gyroscope.filter(
      cb => cb !== callback
    );
  };
}
```

```
onOrientation(callback) {
  this.listeners.orientation.push(callback);
  return () => {
    this.listeners.orientation = this.listeners.orientation.filter(
      cb => cb !== callback
    );
  };
}
```

```
notifyListeners(type, data) {
  this.listeners[type].forEach(callback => {
    try {
      callback(data);
    } catch (error) {
      console.error(`Error in ${type} listener:`, error);
    }
  });
}
```

```
getAccelerometerData() {
  return { ...this.accelerometerData };
}
```

```
getGyroscopeData() {
  return { ...this.gyroscopeData };
}
```

```

}

getOrientationData() {
  return { ...this.orientationData };
}
}

```

Motion-Controlled Game Example:

```

class TiltGame {

  constructor(sensorManager, canvas) {
    this.sensors = sensorManager;
    this.canvas = canvas;
    this.ctx = canvas.getContext('2d');

    this.ball = {
      x: canvas.width / 2,
      y: canvas.height / 2,
      radius: 20,
      vx: 0,
      vy: 0
    };

    this.target = {
      x: 0,
      y: 0,
      radius: 30
    };

    this.score = 0;
    this.running = false;

    this.setupSensors();
  }

  setupSensors() {
    this.sensors.onAccelerometer((data) => {
      this.ball.vx += data.x * 0.5;

```

```
this.ball.vy += data.y * 0.5;
});
}
```

```
start() {
  this.running = true;
  this.placeTarget();
  this.sensors.startAccelerometer(60);
  this.gameLoop();
}
```

```
stop() {
  this.running = false;
  this.sensors.stopAccelerometer();
}
```

```
placeTarget() {
  this.target.x = Math.random() * (this.canvas.width - 60) + 30;
  this.target.y = Math.random() * (this.canvas.height - 60) + 30;
}
```

```
gameLoop() {
  if (!this.running) return;
```

```
  this.update();
  this.draw();
```

```
  requestAnimationFrame(() => this.gameLoop());
}
```

```
update() {
  this.ball.vx *= 0.98;
  this.ball.vy *= 0.98;
```

```
  this.ball.x += this.ball.vx;
  this.ball.y += this.ball.vy;
```

```
  if (this.ball.x - this.ball.radius < 0) {
    this.ball.x = this.ball.radius;
    this.ball.vx *= -0.5;
```

```

}
if (this.ball.x + this.ball.radius > this.canvas.width) {
  this.ball.x = this.canvas.width - this.ball.radius;
  this.ball.vx *= -0.5;
}
if (this.ball.y - this.ball.radius < 0) {
  this.ball.y = this.ball.radius;
  this.ball.vy *= -0.5;
}
if (this.ball.y + this.ball.radius > this.canvas.height) {
  this.ball.y = this.canvas.height - this.ball.radius;
  this.ball.vy *= -0.5;
}

const dx = this.ball.x - this.target.x;
const dy = this.ball.y - this.target.y;
const distance = Math.sqrt(dx * dx + dy * dy);

if (distance < this.ball.radius + this.target.radius) {
  this.score++;
  this.placeTarget();

  if (window.Telegram?.WebApp?.HapticFeedback) {
    window.Telegram.WebApp.HapticFeedback.notificationOccurred('success')
  }
}

draw() {
  this.ctx.fillStyle = 'var(--tg-theme-bg-color)';
  this.ctx.fillRect(0, 0, this.canvas.width, this.canvas.height);

  this.ctx.beginPath();
  this.ctx.arc(this.target.x, this.target.y, this.target.radius, 0, Math.PI
* 2);
  this.ctx.fillStyle = '#4CAF50';
  this.ctx.fill();

  this.ctx.beginPath();
  this.ctx.arc(this.ball.x, this.ball.y, this.ball.radius, 0, Math.PI * 2);

```

```

this.ctx.fillStyle = 'var(--tg-theme-button-color)';
this.ctx.fill();

this.ctx.fillStyle = 'var(--tg-theme-text-color)';
this.ctx.font = '24px Arial';
this.ctx.fillText(` Score: ${this.score}`, 20, 40);
}
}

```

SECTION 5: STORAGE APIS

Mini Apps have two storage options: DeviceStorage for persistent data and SecureStorage for sensitive information.

Storage Manager:

```

class StorageManager {

  constructor(telegramService) {
    this.tg = telegramService.tg;
    this.deviceStorage = this.tg.DeviceStorage;
    this.secureStorage = this.tg.SecureStorage;
    this.cloudStorage = this.tg.CloudStorage;
  }

  isDeviceStorageAvailable() {
    return !!this.deviceStorage;
  }

  isSecureStorageAvailable() {
    return !!this.secureStorage;
  }

  isCloudStorageAvailable() {
    return !!this.cloudStorage;
  }

  async setItem(key, value) {
    if (!this.isDeviceStorageAvailable()) {

```



```
localStorage.setItem(key, JSON.stringify(value));  
return;  
}
```

```
try {  
  await this.deviceStorage.setItem(key, JSON.stringify(value));  
} catch (error) {  
  console.error('DeviceStorage setItem error:', error);  
  localStorage.setItem(key, JSON.stringify(value));  
}  
}
```

```
async getItem(key) {  
  if (!this.isDeviceStorageAvailable()) {  
    const value = localStorage.getItem(key);  
    return value ? JSON.parse(value) : null;  
  }  
}
```

```
try {  
  const value = await this.deviceStorage.getItem(key);  
  return value ? JSON.parse(value) : null;  
} catch (error) {  
  console.error('DeviceStorage getItem error:', error);  
  const value = localStorage.getItem(key);  
  return value ? JSON.parse(value) : null;  
}  
}
```

```
async removeItem(key) {  
  if (!this.isDeviceStorageAvailable()) {  
    localStorage.removeItem(key);  
    return;  
  }  
}
```

```
try {  
  await this.deviceStorage.removeItem(key);  
} catch (error) {  
  console.error('DeviceStorage removeItem error:', error);  
  localStorage.removeItem(key);  
}
```

```
}
```

```
async setSecure(key, value) {  
  if (!this.isSecureStorageAvailable()) {  
    console.warn('SecureStorage not available, falling back to  
DeviceStorage');  
    return this.setItem(`secure_${key}`, value);  
  }  
}
```

```
try {  
  await this.secureStorage.setItem(key, JSON.stringify(value));  
} catch (error) {  
  console.error('SecureStorage setItem error:', error);  
  throw error;  
}  
}
```

```
async getSecure(key) {  
  if (!this.isSecureStorageAvailable()) {  
    return this.getItem(`secure_${key}`);  
  }  
}
```

```
try {  
  const value = await this.secureStorage.getItem(key);  
  return value ? JSON.parse(value) : null;  
} catch (error) {  
  console.error('SecureStorage getItem error:', error);  
  throw error;  
}  
}
```

```
async removeSecure(key) {  
  if (!this.isSecureStorageAvailable()) {  
    return this.removeItem(`secure_${key}`);  
  }  
}
```

```
try {  
  await this.secureStorage.removeItem(key);  
} catch (error) {  
  console.error('SecureStorage removeItem error:', error);  
}
```

```
throw error;
}
}
```

```
async setCloud(key, value) {
  if (!this.isCloudStorageAvailable()) {
    console.warn('CloudStorage not available');
    return;
  }
```

```
  return new Promise((resolve, reject) => {
    this.cloudStorage.setItem(key, JSON.stringify(value), (error) => {
      if (error) reject(error);
      else resolve();
    });
  });
}
```

```
async getCloud(key) {
  if (!this.isCloudStorageAvailable()) {
    return null;
  }
```

```
  return new Promise((resolve, reject) => {
    this.cloudStorage.getItem(key, (error, value) => {
      if (error) reject(error);
      else resolve(value ? JSON.parse(value) : null);
    });
  });
}
```

```
async getCloudKeys() {
  if (!this.isCloudStorageAvailable()) {
    return [];
  }
```

```
  return new Promise((resolve, reject) => {
    this.cloudStorage.getKeys((error, keys) => {
      if (error) reject(error);
      else resolve(keys);
    });
  });
}
```

```
});  
});  
}
```

```
async removeCloud(key) {  
  if (!this.isCloudStorageAvailable()) {  
    return;  
  }
```

```
  return new Promise((resolve, reject) => {  
    this.cloudStorage.removeItem(key, (error) => {  
      if (error) reject(error);  
      else resolve();  
    });  
  });  
}
```

User Preferences Manager:

```
class UserPreferences {  
  
  constructor(storageManager) {  
    this.storage = storageManager;  
    this.preferences = null;  
    this.defaultPreferences = {  
      currency: 'USD',  
      language: 'en',  
      notifications: true,  
      theme: 'auto',  
      slippage: 0.5,  
      gasPreset: 'standard',  
      confirmTrades: true,  
      showBalance: true  
    };  
  }  
}
```

```
async load() {  
  const saved = await this.storage.getItem('user_preferences');
```

```
this.preferences = { ...this.defaultPreferences, ...saved };  
return this.preferences;  
}
```

```
async save() {  
  await this.storage.setItem('user_preferences', this.preferences);  
}
```

```
get(key) {  
  if (!this.preferences) {  
    return this.defaultPreferences[key];  
  }  
  return this.preferences[key];  
}
```

```
async set(key, value) {  
  if (!this.preferences) {  
    await this.load();  
  }  
  this.preferences[key] = value;  
  await this.save();  
}
```

```
async setMultiple(updates) {  
  if (!this.preferences) {  
    await this.load();  
  }  
  Object.assign(this.preferences, updates);  
  await this.save();  
}
```

```
async reset() {  
  this.preferences = { ...this.defaultPreferences };  
  await this.save();  
}
```

```
getAll() {  
  return { ...this.preferences };  
}  
}
```

SECTION 6: TON CONNECT INTEGRATION

TON Connect links Mini Apps to user wallets. This is mandatory for any blockchain functionality in Telegram Mini Apps.

TON Connect Service:

```
import TonConnect from '@tonconnect/sdk';
```

```
class TONConnectService {
```

```
  constructor(manifestUrl) {  
    this.manifestUrl = manifestUrl;  
    this.connector = null;  
    this.wallet = null;  
    this.onStatusChangeCallbacks = [];  
  }
```

```
  async init() {  
    this.connector = new TonConnect({  
      manifestUrl: this.manifestUrl  
    });
```

```
    this.connector.onStatusChange((wallet) => {  
      this.wallet = wallet;  
      this.notifyStatusChange(wallet);  
    });
```

```
    await this.connector.restoreConnection();
```

```
    return this.isConnected();  
  }
```

```
  isConnected() {  
    return this.connector?.connected || false;  
  }
```

```
  getWallet() {
```

```
if (!this.connector?.wallet) return null;
```

```
return {  
  address: this.connector.wallet.account.address,  
  chain: this.connector.wallet.account.chain,  
  publicKey: this.connector.wallet.account.publicKey,  
  name: this.connector.wallet.device.appName,  
  platform: this.connector.wallet.device.platform  
};  
}
```

```
getAddress() {  
  return this.connector?.wallet?.account?.address;  
}
```

```
async getWalletsList() {  
  const wallets = await this.connector.getWallets();  
  return wallets.map(wallet => ({  
    name: wallet.name,  
    appName: wallet.appName,  
    imageUrl: wallet.imageUrl,  
    aboutUrl: wallet.aboutUrl,  
    bridgeUrl: wallet.bridgeUrl,  
    universalLink: wallet.universalLink,  
    jsBridgeKey: wallet.jsBridgeKey  
  }));  
}
```

```
async connect(walletName = 'Tonkeeper') {  
  const wallets = await this.connector.getWallets();  
  const wallet = wallets.find(w =>  
    w.name.toLowerCase() === walletName.toLowerCase() ||  
    w.appName.toLowerCase() === walletName.toLowerCase()  
  );
```

```
if (!wallet) {  
  throw new Error(`Wallet ${walletName} not found`);  
}
```

```
const universalLink = this.connector.connect({
```

```
bridgeUrl: wallet.bridgeUrl,  
universalLink: wallet.universalLink  
});
```

```
return universalLink;  
}
```

```
async disconnect() {  
  await this.connector.disconnect();  
  this.wallet = null;  
}
```

```
async sendTransaction(params) {  
  if (!this.isConnected()) {  
    throw new Error('Wallet not connected');  
  }
```

```
  const transaction = {  
    validUntil: Math.floor(Date.now() / 1000) + 600,  
    messages: params.messages.map(msg => ({  
      address: msg.address,  
      amount: msg.amount.toString(),  
      payload: msg.payload,  
      stateInit: msg.stateInit  
    })))  
  };

```

```
  const result = await this.connector.sendTransaction(transaction);  
  return result;  
}
```

```
async sendTON(recipient, amountNano, comment = "") {  
  const messages = [{  
    address: recipient,  
    amount: amountNano.toString()  
  }];
```

```
  if (comment) {  
    const encoder = new TextEncoder();  
    const commentBytes = encoder.encode(comment);
```



```
const payload = this.createCommentPayload(commentBytes);
messages[0].payload = payload;
}
```

```
return this.sendTransaction({ messages });
}
```

```
createCommentPayload(comment) {
return Buffer.from([0, ...comment]).toString('base64');
}
```

```
onStatusChange(callback) {
this.onStatusChangeCallbacks.push(callback);
return () => {
this.onStatusChangeCallbacks =
this.onStatusChangeCallbacks.filter(
cb => cb !== callback
);
};
}
```

```
notifyStatusChange(wallet) {
this.onStatusChangeCallbacks.forEach(callback => {
try {
callback(wallet);
} catch (error) {
console.error('Error in status change callback:', error);
}
});
}
}
```

TON Connect Manifest:

Create tonconnect-manifest.json in your public folder:

```
{
"url": "https://your-mini-app.com",
"name": "Crypto Mini App",
```

```
"iconUrl": "https://your-mini-app.com/icon.png",
"termsOfUseUrl": "https://your-mini-app.com/terms",
"privacyPolicyUrl": "https://your-mini-app.com/privacy"
}
```

Wallet Connection UI Component:

```
class WalletConnector {

  constructor(tonConnect, telegramService) {
    this.ton = tonConnect;
    this.tg = telegramService;
    this.container = null;
  }

  async render(containerId) {
    this.container = document.getElementById(containerId);

    if (this.ton.isConnected()) {
      this.renderConnectedState();
    } else {
      this.renderDisconnectedState();
    }

    this.ton.onStatusChange((wallet) => {
      if (wallet) {
        this.renderConnectedState();
      } else {
        this.renderDisconnectedState();
      }
    });
  }

  renderConnectedState() {
    const wallet = this.ton.getWallet();
    const shortAddress = `${wallet.address.slice(0, 6)}...
    ${wallet.address.slice(-4)}`;

    this.container.innerHTML = `
```

```

<div class="wallet-card connected">
<div class="wallet-info">
<div class="wallet-icon"> W </div>
<div class="wallet-details">
<div class="wallet-name"> ${wallet.name} </div>
<div class="wallet-address"> ${shortAddress} </div>
</div>
</div>
<button class="disconnect-btn"
id="disconnectBtn"> Disconnect </button>
</div>
`;

```

```

document.getElementById('disconnectBtn').addEventListener('click',
() => {
this.handleDisconnect();
});
}

```

```

renderDisconnectedState() {
this.container.innerHTML = `
<div class="wallet-card disconnected">
<div class="wallet-prompt">
<h3> Connect Wallet </h3>
<p class="hint"> Connect your TON wallet to use this app </p>
</div>
<button class="connect-btn button" id="connectBtn"> Connect
Wallet </button>
</div>
`;

```

```

document.getElementById('connectBtn').addEventListener('click', ()
=> {
this.handleConnect();
});
}

```

```

async handleConnect() {
try {
const wallets = await this.ton.getWalletsList();

```

```
this.showWalletSelector(wallets);
} catch (error) {
  console.error('Failed to get wallets:', error);
  this.tg.showAlert('Failed to load wallets. Please try again.');
```

```
}
}

showWalletSelector(wallets) {
  const buttons = wallets.slice(0, 3).map(wallet => ({
    id: wallet.appName,
    type: 'default',
    text: wallet.name
  }));
```

```
    buttons.push({
      id: 'cancel',
      type: 'cancel',
      text: 'Cancel'
    });
```

```
    this.tg.showPopup({
      title: 'Select Wallet',
      message: 'Choose a wallet to connect',
      buttons: buttons
    }).then(async (buttonId) => {
      if (buttonId === 'cancel') return;
```

```
      await this.connectToWallet(buttonId);
    });
  }
```

```
  async connectToWallet(walletAppName) {
    try {
      const universalLink = await this.ton.connect(walletAppName);
```

```
      const platform = this.tg.getPlatform();
```

```
      if (platform === 'ios' || platform === 'android') {
        window.location.href = universalLink;
```

```

    } else {
      this.showQRCode(universalLink);
    }
  } catch (error) {
    console.error('Failed to connect:', error);
    this.tg.showAlert('Failed to connect wallet. Please try again.');
```

```

  }
}

showQRCode(link) {
  console.log('Show QR for link:', link);
}
```

```

async handleDisconnect() {
  const confirmed = await this.tg.showConfirm('Disconnect
wallet?');
```

```

  if (confirmed) {
    await this.ton.disconnect();
  }
}
}
```

SECTION 7: PAYMENT INTEGRATION

Mini Apps can process payments through Telegram Stars using the `openInvoice` method.

Payment Service:

```

class PaymentService {

  constructor(telegramService, apiService) {
    this.tg = telegramService;
    this.api = apiService;
  }

  async createInvoice(item) {
    const response = await this.api.post('/payments/create-invoice', {
```

```
item_id: item.id,  
item_name: item.name,  
item_description: item.description,  
stars_amount: item.price  
});
```

```
return response.invoice_link;  
}
```

```
async openPayment(item) {  
  try {  
    const invoiceLink = await this.createInvoice(item);  
  
    const status = await this.tg.openInvoice(invoiceLink);  
  
    return this.handlePaymentResult(status, item);  
  } catch (error) {  
    console.error('Payment error:', error);  
    throw error;  
  }  
}
```

```
handlePaymentResult(status, item) {  
  switch (status) {  
    case 'paid':  
      return {  
        success: true,  
        message: `Successfully purchased ${item.name}!`  
      };  

```

```
    case 'cancelled':  
      return {  
        success: false,  
        message: 'Payment was cancelled'  
      };  

```

```
    case 'failed':  
      return {  
        success: false,  
        message: 'Payment failed. Please try again.'  
      };  

```

```
};
```

```
case 'pending':  
  return {  
    success: false,  
    pending: true,  
    message: 'Payment is being processed...'  
  };  
};
```

```
default:  
  return {  
    success: false,  
    message: `Unknown payment status: ${status}`  
  };  
}  
}
```

```
async createSubscription(plan) {  
  const response = await this.api.post('/payments/create-  
subscription', {  
    plan_id: plan.id,  
    plan_name: plan.name,  
    stars_amount: plan.price,  
    period_seconds: plan.period  
  });  
};
```

```
return response.invoice_link;  
}
```

```
async openSubscription(plan) {  
  try {  
    const invoiceLink = await this.createSubscription(plan);  
    const status = await this.tg.openInvoice(invoiceLink);  
    return this.handlePaymentResult(status, plan);  
  } catch (error) {  
    console.error('Subscription error:', error);  
    throw error;  
  }  
}  
}
```

Shop Component:

```
class ShopComponent {

  constructor(paymentService, telegramService) {
    this.payments = paymentService;
    this.tg = telegramService;
    this.items = [];
  }

  setItems(items) {
    this.items = items;
  }

  render(containerId) {
    const container = document.getElementById(containerId);

    container.innerHTML = `
    <div class="shop-header">
    <h2>Premium Features </h2>
    <p class="hint">Purchase with Telegram Stars </p>
    </div>
    <div class="shop-items" id="shopItems"> </div>
    `;

    const itemsContainer = document.getElementById('shopItems');

    this.items.forEach(item => {
      const itemEl = this.createItemElement(item);
      itemsContainer.appendChild(itemEl);
    });
  }

  createItemElement(item) {
    const div = document.createElement('div');
    div.className = 'shop-item card';
    div.innerHTML = `
    <div class="item-info">
```



```

<div class="item-icon">${item.icon}</div>
<div class="item-details">
<div class="item-name">${item.name}</div>
<div class="item-description hint">${item.description}</div>
</div>
</div>
<button class="buy-btn button" data-item-id="${item.id}">
${item.price} Stars
</button>
`;

```

```

div.querySelector('.buy-btn').addEventListener('click', () => {
  this.handlePurchase(item);
});

```

```

return div;
}

```

```

async handlePurchase(item) {
  this.tg.hapticFeedback('impact', 'light');

```

```

try {
  const result = await this.payments.openPayment(item);

  if (result.success) {
    this.tg.hapticFeedback('notification', 'success');
    await this.tg.showAlert(result.message);
    this.onPurchaseComplete(item);
  } else if (result.message) {
    if (!result.pending) {
      this.tg.hapticFeedback('notification', 'error');
    }
    await this.tg.showAlert(result.message);
  }
} catch (error) {
  this.tg.hapticFeedback('notification', 'error');
  await this.tg.showAlert('Payment failed. Please try again.');
```

```
onPurchaseComplete(item) {  
  }  
}
```

SECTION 8: SHARING AND VIRALITY

Mini Apps can share content to chats and stories, enabling viral distribution.

Share Service:

```
class ShareService {  
  
  constructor(telegramService) {  
    this.tg = telegramService;  
  }  
  
  shareToChat(text, url = null) {  
    let shareUrl = `https://t.me/share/url?text=${encodeURIComponent(text)}`;  
  
    if (url) {  
      shareUrl += `&url=${encodeURIComponent(url)}`;  
    }  
  
    this.tg.openTelegramLink(shareUrl);  
  }  
  
  shareReferrallink(referralCode) {  
    const botUsername = 'your_bot';  
    const referrallink = `https://t.me/${botUsername}?start=ref_${referralCode}`;  
  
    const text = `Join me on CryptoBot and get 100 bonus Stars!\n\n${referrallink}`;  
  
    this.shareToChat(text);  
  }  
}
```

```
async shareToStory(mediaUrl, text) {  
  if (!this.tg.isVersionAtLeast('8.0')) {  
    console.warn('Story sharing not supported');  
    return false;  
  }
```

```
  try {  
    await this.tg.shareToStory(mediaUrl, {  
      text: text,  
      widget_link: {  
        url: 'https://t.me/your_bot/app',  
        name: 'Open App'  
      }  
    });  
    return true;  
  } catch (error) {  
    console.error('Story sharing failed:', error);  
    return false;  
  }  
}
```

```
shareTradeResult(trade) {  
  const text = `I just made ${trade.profit > 0 ? '+' :  
  }${trade.profit}% on ${trade.token}`;
```

```
  this.shareToChat(text);  
}
```

```
sharePortfolioPerformance(performance) {  
  const text = `My portfolio is ${performance > 0 ? 'up' : 'down'}  
  ${Math.abs(performance)}% this week!`;
```

```
  this.shareToChat(text);  
}
```

```
shareAchievement(achievement) {  
  const text = `I just unlocked the "${achievement.name}"  
  achievement in CryptoBot!`;
```

```
  this.shareToChat(text);
```

```

}

generateReferralCode(userId) {
  const encoded = btoa(String(userId));
  return encoded.slice(0, 8);
}
}

```

SECTION 9: COMPLETE MINI APP EXAMPLE

Putting it all together - a complete trading Mini App:

```

import telegramService from './services/telegram.js';
import { TONConnectService } from './services/ton.js';
import { StorageManager } from './services/storage.js';
import { PaymentService } from './services/payment.js';
import { SensorManager } from './services/sensors.js';
import { FullScreenManager } from './services/fullscreen.js';
import { ShareService } from './services/share.js';
import { APIService } from './services/api.js';

```

```

class CryptoMiniApp {

  constructor() {
    this.tg = telegramService;
    this.ton = null;
    this.storage = null;
    this.payments = null;
    this.sensors = null;
    this.fullscreen = null;
    this.share = null;
    this.api = null;

    this.user = null;
    this.preferences = null;
    this.currentPage = 'home';
  }

```

```

  async init() {

```

```
if (!this.tg.init()) {
  this.showError("This app must be opened in Telegram");
  return;
}

this.user = this.tg.getUser();

this.api = new APIService('https://api.your-mini-app.com');
this.api.setAuthToken(this.tg.getInitData());

this.storage = new StorageManager(this.tg);
this.preferences = await this.loadPreferences();

this.ton = new TONConnectService('/tonconnect-manifest.json');
await this.ton.init();

this.payments = new PaymentService(this.tg, this.api);

this.sensors = new SensorManager(this.tg);

this.fullscreen = new FullScreenManager(this.tg);

this.share = new ShareService(this.tg);

this.setupNavigation();

this.render();

this.tg.showMainButton('Connect Wallet', () => {
  this.handleMainButtonClick();
});

this.updateMainButton();

console.log('Mini App initialized');
console.log('User:', this.user);
console.log('Wallet connected:', this.ton.isConnected());
}

async loadPreferences() {
```

```
const saved = await this.storage.getItem('preferences');
return saved || {
  currency: 'USD',
  notifications: true,
  slippage: 0.5
};
}
```

```
setupNavigation() {
  this.tg.onBackButtonClicked = () => {
    this.navigateBack();
  };
}
```

```
this.tg.onMainButtonClicked = () => {
  this.handleMainButtonClick();
};
```

```
this.ton.onStatusChange((wallet) => {
  this.updateMainButton();
  this.render();
});
}
```

```
updateMainButton() {
  if (this.ton.isConnected()) {
    const wallet = this.ton.getWallet();
    const shortAddress = `${wallet.address.slice(0, 4)}...
    ${wallet.address.slice(-4)}`;
    this.tg.showMainButton(shortAddress);
  } else {
    this.tg.showMainButton('Connect Wallet');
  }
}
```

```
handleMainButtonClick() {
  if (this.ton.isConnected()) {
    this.showWalletOptions();
  } else {
    this.connectWallet();
  }
}
```

```

}

async connectWallet() {
  try {
    const wallets = await this.ton.getWalletsList();

    if (wallets.length === 0) {
      await this.tg.showAlert('No wallets available');
      return;
    }

    const link = await this.ton.connect('Tonkeeper');

    const platform = this.tg.getPlatform();
    if (platform === 'ios' || platform === 'android') {
      window.location.href = link;
    }
  } catch (error) {
    console.error('Connect error:', error);
    await this.tg.showAlert('Failed to connect wallet');
  }
}

async showWalletOptions() {
  const buttonId = await this.tg.showPopup({
    title: 'Wallet',
    message: this.ton.getAddress(),
    buttons: [
      { id: 'copy', type: 'default', text: 'Copy Address' },
      { id: 'disconnect', type: 'destructive', text: 'Disconnect' },
      { id: 'cancel', type: 'cancel', text: 'Cancel' }
    ]
  });

  if (buttonId === 'copy') {
    await navigator.clipboard.writeText(this.ton.getAddress());
    this.tg.hapticFeedback('notification', 'success');
  } else if (buttonId === 'disconnect') {
    await this.ton.disconnect();
  }
}

```

```
}
```

```
render() {  
  const app = document.getElementById('app');
```

```
  app.innerHTML = `
```

```
    <div class="app-container">
```

```
      <header class="app-header">
```

```
        <h1>CryptoBot</h1>
```

```
        <div class="user-greeting">Hello, ${this.user?.first_name ||  
'User'}</div>
```

```
      </header>
```

```
      <main class="app-content" id="pageContent">
```

```
        ${this.renderPage()}</main>
```

```
    <nav class="bottom-nav">
```

```
      <button class="nav-item ${this.currentPage === 'home' ?  
'active' : ""}
```

```
data-page="home">Home</button>
```

```
      <button class="nav-item ${this.currentPage === 'trade' ?  
'active' : ""}
```

```
data-page="trade">Trade</button>
```

```
      <button class="nav-item ${this.currentPage === 'wallet' ?  
'active' : ""}
```

```
data-page="wallet">Wallet</button>
```

```
      <button class="nav-item ${this.currentPage === 'settings' ?  
'active' : ""}
```

```
data-page="settings">Settings</button>
```

```
    </nav>
```

```
  </div>
```

```
`;  
};
```

```
this.attachEventListeners();
```

```
}
```

```
renderPage() {
```

```
  switch (this.currentPage) {
```

```
    case 'home':
```



```

return this.renderHomePage();
case 'trade':
return this.renderTradePage();
case 'wallet':
return this.renderWalletPage();
case 'settings':
return this.renderSettingsPage();
default:
return this.renderHomePage();
}
}

```

```

renderHomePage() {
return `
<div class="home-page">
<div class="portfolio-summary card">
<div class="portfolio-value"> $12,345.67 </div>
<div class="portfolio-change positive"> + 5.67% today </div>
</div>

<div class="quick-actions">
<button class="action-btn" data-action="buy"> Buy </button>
<button class="action-btn" data-action="sell"> Sell </button>
<button class="action-btn" data-action="swap"> Swap </
button>
</div>

```

```

<div class="market-overview">
<h3> Market </h3>
<div class="token-list">
${this.renderTokenList()}
</div>
</div>
</div>
`;
}

```

```

renderTokenList() {
const tokens = [
{ symbol: 'TON', price: 5.67, change: 5.8 },

```

```
{ symbol: 'BTC', price: 43210, change: 2.3 },
{ symbol: 'ETH', price: 2345, change: -1.2 }
];
```

```
return tokens.map(token => `
  <div class="token-item card" data-token="${token.symbol}">
    <div class="token-info">
      <div class="token-symbol">${token.symbol}</div>
      <div class="token-price">${token.price.toLocaleString()}</
div>
</div>
<div class="token-change ${token.change >= 0 ? 'positive' :
'negative'}">
  ${token.change >= 0 ? '+' : ''}${token.change}%
</div>
</div>
`);
}
```

```
renderTradePage() {
  return `
    <div class="trade-page">
      <h2>Trade</h2>
      <div class="trade-form card">
        <div class="form-group">
          <label>From</label>
          <select id="fromToken">
            <option value="TON">TON</option>
            <option value="USDT">USDT</option>
          </select>
          <input type="number" id="fromAmount" placeholder="0.00">
        </div>

        <div class="swap-icon">swap</div>

        <div class="form-group">
          <label>To</label>
          <select id="toToken">
            <option value="USDT">USDT</option>
            <option value="TON">TON</option>
          </select>
        </div>
      </div>
    </div>
  `;
}
```

```

</select>
<input type="number" id="toAmount" placeholder="0.00"
readonly>
</div>

<button class="button swap-btn" id="executeSwap">Swap</
button>
</div>
</div>
`;
}

```

```

renderWalletPage() {
if (!this.ton.isConnected()) {
return `
<div class="wallet-page">
<div class="connect-prompt card">
<h3>Connect Wallet</h3>
<p class="hint">Connect your TON wallet to view your
balance</p>
<button class="button" id="connectWalletBtn">Connect</
button>
</div>
</div>
`;
}
}

```

```
const wallet = this.ton.getWallet();
```

```

return `
<div class="wallet-page">
<div class="wallet-info card">
<div class="wallet-address">${wallet.address.slice(0, 8)}...
${wallet.address.slice(-6)}</div>
<div class="wallet-name">${wallet.name}</div>
</div>

<div class="balance-list">
<div class="balance-item card">
<div class="token-icon">T</div>

```

```

<div class="token-info">
<div class="token-name"> Toncoin </div>
<div class="token-amount"> 100.00 TON </div>
</div>
<div class="token-value"> $567.00 </div>
</div>
</div>

<div class="wallet-actions">
<button class="button" id="sendBtn"> Send </button>
<button class="button" id="receiveBtn"> Receive </button>
</div>
</div>
`;
}

```

```

renderSettingsPage() {
return `
<div class="settings-page">
<h2> Settings </h2>

<div class="settings-list">
<div class="settings-item card">
<div class="setting-label"> Currency </div>
<div class="setting-value"> ${this.preferences.currency} </div>
</div>

<div class="settings-item card">
<div class="setting-label"> Notifications </div>
<div class="setting-value"> ${this.preferences.notifications ? 'ON'
: 'OFF'} </div>
</div>

<div class="settings-item card">
<div class="setting-label"> Slippage </div>
<div class="setting-value"> ${this.preferences.slippage}% </
div>
</div>
</div>

```

```
<button class="button share-btn" id="shareBtn"> Share App</button>
</div>
`
}
```

```
attachEventListeners() {
  document.querySelectorAll('.nav-item').forEach(btn => {
    btn.addEventListener('click', (e) => {
      const page = e.target.dataset.page;
      this.navigate(page);
    });
  });
}
```

```
const connectBtn = document.getElementById('connectWalletBtn');
if (connectBtn) {
  connectBtn.addEventListener('click', () => this.connectWallet());
}
```

```
const shareBtn = document.getElementById('shareBtn');
if (shareBtn) {
  shareBtn.addEventListener('click', () => {
    this.share.shareReferralLink(this.user.id);
  });
}
```

```
const swapBtn = document.getElementById('executeSwap');
if (swapBtn) {
  swapBtn.addEventListener('click', () => this.executeSwap());
}
}
```

```
navigate(page) {
  this.currentPage = page;

  if (page !== 'home') {
    this.tg.showBackButton();
  } else {
    this.tg.hideBackButton();
  }
}
```

```
this.tg.hapticFeedback('selection');
this.render();
}
```

```
navigateBack() {
  this.navigate('home');
}
```

```
async executeSwap() {
  if (!this.ton.isConnected()) {
    await this.connectWallet();
    return;
  }
```

```
this.tg.setMainButtonLoading(true);
```

```
await new Promise(resolve => setTimeout(resolve, 2000));
```

```
this.tg.setMainButtonLoading(false);
this.tg.hapticFeedback('notification', 'success');
await this.tg.showAlert('Swap executed successfully!');
}
```

```
showError(message) {
  document.getElementById('app').innerHTML = `
    <div class="error-page">
      <h2>Error</h2>
      <p>${message}</p>
    </div>
  `;
}
```

```
document.addEventListener('DOMContentLoaded', () => {
  const app = new CryptoMiniApp();
  app.init();
});
```

SECTION 10: DEPLOYING MINI APPS

Deploy configuration and bot setup for Mini Apps.

Vite Configuration:

```
import { defineConfig } from 'vite';

export default defineConfig({
  base: './',
  build: {
    outDir: 'dist',
    assetsDir: 'assets',
    sourcemap: false,
    minify: 'terser'
  },
  server: {
    port: 3000,
    host: true
  }
});
```

Bot Setup for Mini App:

```
from telegram import Update, WebAppInfo, InlineKeyboardButton,
InlineKeyboardMarkup
from telegram.ext import Application, CommandHandler

class MiniAppBot:

    def __init__(self, token, webapp_url):
        self.token = token
        self.webapp_url = webapp_url
        self.application = Application.builder().token(token).build()
        self.setup_handlers()

    def setup_handlers(self):
        self.application.add_handler(
            CommandHandler("start", self.cmd_start)
```

```

)
self.application.add_handler(
CommandHandler("app", self.cmd_app)
)

async def cmd_start(self, update: Update, context):
keyboard = InlineKeyboardMarkup([
    InlineKeyboardButton(
        "Open App",
        web_app=WebAppInfo(url=self.webapp_url)
    )
])

await update.message.reply_text(
    "Welcome! Tap below to open the Mini App.",
    reply_markup=keyboard
)

async def cmd_app(self, update: Update, context):
keyboard = InlineKeyboardMarkup([
    InlineKeyboardButton(
        "Open Mini App",
        web_app=WebAppInfo(url=self.webapp_url)
    )
])

await update.message.reply_text(
    "Tap to launch:",
    reply_markup=keyboard
)

def run(self):
self.application.run_polling()

```

CHAPTER SUMMARY

This chapter covered Mini Apps development:

Mini Apps are web applications in Telegram's WebView. Standard

HTML, CSS, JavaScript with Telegram-specific APIs.

The Telegram Web App API provides user data, theme information, navigation controls, and payment integration.

Full-screen mode enables immersive experiences. Safe area insets prevent content hiding behind system UI.

Device sensors open new possibilities. Accelerometer, gyroscope, and orientation enable motion-controlled games.

Storage APIs persist data. DeviceStorage for preferences, SecureStorage for sensitive data, CloudStorage for sync across devices.

TON Connect is mandatory for wallet integration. The exclusive protocol for Telegram Mini Apps blockchain features.

Telegram Stars power payments. openInvoice handles the payment flow. Backend creates invoices via Bot API.

Sharing enables virality. Chat sharing, story posts, and referral links spread your app.

Next chapter: Telegram Stars and Payments. We dive deep into monetization infrastructure.

CHAPTER 4: TELEGRAM STARS AND PAYMENTS

The complete monetization infrastructure for Telegram bots and Mini Apps. Telegram Stars (XTR) are mandatory for digital goods. This chapter covers invoice creation, subscription management, payment processing, and withdrawing earnings to TON.

SECTION 1: UNDERSTANDING TELEGRAM STARS

Telegram Stars are the virtual currency powering Telegram's digital economy. Users buy Stars through in-app purchases (Apple Pay, Google Pay) and spend them in bots and Mini Apps. Developers earn Stars and withdraw them to Toncoin.

Why Stars Exist:

Apple and Google require in-app purchases for digital goods sold in mobile apps. They take 30% of revenue. Telegram negotiated a solution: Telegram Stars bypass direct app store payment requirements while remaining compliant. The result is lower fees and seamless UX.

The Star Economy:

Users purchase Stars at approximately \$0.013 per Star (rates vary by region and platform). They spend Stars on digital goods, premium features, subscriptions, and tips. Developers receive Stars minus Telegram's cut (approximately 15-20%).

What Can Be Sold for Stars:

- Digital goods and services
- Premium features and upgrades
- Subscription access
- In-app currencies and items
- Digital content and media
- Bot functionality and automation
- Mini App features

What Cannot Be Sold for Stars:

- Physical goods (use external payment providers)
- Real-world services (use external payment)
- Gambling or lottery tickets
- Anything violating Telegram's terms

Currency Tag:

Stars use the currency code "XTR" in the Bot API. All star amounts are in whole Stars (no decimals). Minimum transaction is typically 1 Star.

SECTION 2: CREATING STAR INVOICES

Invoices are the payment requests users see. Create them via the Bot API, then send or share them for payment.

Basic Invoice Creation:

```
from telegram import Update, LabeledPrice
from telegram.ext import (
    Application, CommandHandler, PreCheckoutQueryHandler,
    MessageHandler, filters, ContextTypes
)

class StarPaymentBot:

    def __init__(self, token):
        self.token = token
        self.application = Application.builder().token(token).build()
        self.setup_handlers()

    def setup_handlers(self):
        self.application.add_handler(
            CommandHandler("buy", self.cmd_buy)
        )
        self.application.add_handler(
            CommandHandler("premium", self.cmd_premium)
        )
        self.application.add_handler(
            PreCheckoutQueryHandler(self.pre_checkout)
        )
        self.application.add_handler(
            MessageHandler(
                filters.SUCCESSFUL_PAYMENT,
                self.successful_payment
            )
        )

    async def cmd_buy(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
        prices = [LabeledPrice(label="Premium Access", amount=100)]
```

```
await update.message.reply_invoice(
    title="Premium Access",
    description="Unlock all premium features for 30 days",
    payload="premium_30_days",
    currency="XTR",
    prices=prices
)
```

```
async def cmd_premium(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
```

```
    tiers = [
        {
            "name": "Basic",
            "stars": 50,
            "features": "Price alerts, Portfolio view"
        },
        {
            "name": "Pro",
            "stars": 200,
            "features": "All Basic + Trading, Analytics"
        },
        {
            "name": "Whale",
            "stars": 1000,
            "features": "All Pro + Priority support, API access"
        }
    ]
```

```
    for tier in tiers:
```

```
        prices = [LabeledPrice(label=tier["name"],
amount=tier["stars"])]
```

```
await update.message.reply_invoice(
    title=f'{tier["name"]} Plan',
    description=tier["features"],
    payload=f'tier_{tier["name"].lower()}',
    currency="XTR",
    prices=prices
)
```

```
async def pre_checkout(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
    query = update.pre_checkout_query
```

```
    payload = query.invoice_payload
    user_id = query.from_user.id
    total_amount = query.total_amount
```

```
    is_valid = await self.validate_purchase(payload, user_id,
total_amount)
```

```
    if is_valid:
        await query.answer(ok=True)
    else:
        await query.answer(
            ok=False,
            error_message="This purchase is no longer available"
        )
```

```
async def validate_purchase(self, payload, user_id, amount):
    valid_payloads = {
        "premium_30_days": 100,
        "tier_basic": 50,
        "tier_pro": 200,
        "tier_whale": 1000
    }
```

```
    expected_amount = valid_payloads.get(payload)
```

```
    if expected_amount is None:
        return False
```

```
    if amount != expected_amount:
        return False
```

```
    return True
```

```
async def successful_payment(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
```

```
payment = update.message.successful_payment
```

```
user_id = update.effective_user.id
```

```
payload = payment.invoice_payload
```

```
amount = payment.total_amount
```

```
charge_id = payment.telegram_payment_charge_id
```

```
await self.process_purchase(user_id, payload, amount, charge_id)
```

```
await update.message.reply_text(
```

```
f"Payment successful!\n\n"
```

```
f"You paid {amount} Stars.\n"
```

```
f"Your purchase has been activated.\n\n"
```

```
f"Transaction ID: {charge_id[:16]}..."
```

```
)
```

```
async def process_purchase(self, user_id, payload, amount,  
charge_id):
```

```
print(f"Processing: {payload} for user {user_id}")
```

```
print(f"Amount: {amount} Stars")
```

```
print(f"Charge ID: {charge_id}")
```

```
def run(self):
```

```
self.application.run_polling()
```

SECTION 3: INVOICE LINKS

Invoice links allow payments without an active conversation. Share them anywhere - in groups, websites, or other apps.

Creating Invoice Links:

```
class InvoiceLinkGenerator:
```

```
def __init__(self, bot):
```

```
self.bot = bot
```

```
async def create_product_link(self, product):
```

```
prices = [LabeledPrice(label=product["name"],
```

```
amount = product["stars"])]
```

```
link = await self.bot.create_invoice_link(  
title = product["name"],  
description = product["description"],  
payload = f"product:{product['id']}",  
currency = "XTR",  
prices = prices  
)
```

```
return link
```

```
async def create_donation_link(self, suggested_amounts):  
links = []
```

```
for amount in suggested_amounts:  
prices = [LabeledPrice(label=f'Donate {amount} Stars',  
amount = amount)]
```

```
link = await self.bot.create_invoice_link(  
title=f'Donate {amount} Stars',  
description="Support the developer",  
payload=f'donation:{amount}',  
currency="XTR",  
prices = prices  
)
```

```
links.append({"amount": amount, "link": link})
```

```
return links
```

```
async def create_tip_link(self, content_id, creator_name):  
prices = [LabeledPrice(label="Tip", amount = 10)]
```

```
link = await self.bot.create_invoice_link(  
title=f'Tip {creator_name}',  
description=f'Say thanks for great content',  
payload=f'tip:{content_id}',  
currency="XTR",  
prices = prices
```

)

return link

```
async def create_unlock_link(self, content_id, content_title, price):  
    prices = [LabeledPrice(label="Unlock", amount=price)]
```

```
    link = await self.bot.create_invoice_link(  
        title=f"Unlock: {content_title}",  
        description="Get full access to this content",  
        payload=f"unlock:{content_id}",  
        currency="XTR",  
        prices=prices  
    )
```

return link

Using Invoice Links in Mini Apps:

```
class MiniAppPayments {
```

```
    constructor(telegramService, apiService) {  
        this.tg = telegramService;  
        this.api = apiService;  
    }
```

```
    async purchaseItem(itemId) {  
        const response = await this.api.post('/payments/create-link', {  
            item_id: itemId  
        });
```

```
        const invoiceLink = response.invoice_link;
```

```
        const status = await this.openInvoice(invoiceLink);
```

```
        return this.handlePaymentStatus(status);  
    }
```

```
    openInvoice(link) {
```



```
return new Promise((resolve) => {  
  this.tg.tg.openInvoice(link, (status) => {  
    resolve(status);  
  });  
});  
}
```

```
handlePaymentStatus(status) {  
  switch (status) {  
    case 'paid':  
      this.tg.hapticFeedback('notification', 'success');  
      return { success: true, status: 'paid' };  

```

```
    case 'cancelled':  
      return { success: false, status: 'cancelled' };  

```

```
    case 'failed':  
      this.tg.hapticFeedback('notification', 'error');  
      return { success: false, status: 'failed' };  

```

```
    case 'pending':  
      return { success: false, status: 'pending' };  

```

```
    default:  
      return { success: false, status: 'unknown' };  
  }  
}
```

```
async purchaseSubscription(planId) {  
  const response = await this.api.post('/payments/create-  
subscription-link', {  
    plan_id: planId  
  });  

```

```
  const status = await this.openInvoice(response.invoice_link);
```

```
  return this.handlePaymentStatus(status);  
}  
}
```

SECTION 4: SUBSCRIPTION PAYMENTS

Subscriptions automatically renew, charging users periodically. This enables recurring revenue from premium features.

Creating Subscriptions:

```
class SubscriptionManager:
```

```
    def __init__(self, bot, database):
```

```
        self.bot = bot
```

```
        self.db = database
```

```
        self.plans = {
```

```
            "monthly_basic": {
```

```
                "name": "Basic Monthly",
```

```
                "stars": 100,
```

```
                "period": 2592000,
```

```
                "features": ["price_alerts", "portfolio"]
```

```
            },
```

```
            "monthly_pro": {
```

```
                "name": "Pro Monthly",
```

```
                "stars": 500,
```

```
                "period": 2592000,
```

```
                "features": ["price_alerts", "portfolio", "trading", "analytics"]
```

```
            },
```

```
            "yearly_pro": {
```

```
                "name": "Pro Yearly",
```

```
                "stars": 5000,
```

```
                "period": 31536000,
```

```
                "features": ["price_alerts", "portfolio", "trading", "analytics",
```

```
                "priority_support"]
```

```
            }
```

```
        }
```

```
    async def create_subscription_link(self, plan_id):
```

```
        plan = self.plans.get(plan_id)
```

```
        if not plan:
```

```
raise ValueError(f"Unknown plan: {plan_id}")
```

```
prices = [LabeledPrice(label = plan["name"],  
amount = plan["stars"])]
```

```
link = await self.bot.create_invoice_link(  
title = plan["name"],  
description = f'Features: {', '.join(plan["features"])}',  
payload = f'subscription:{plan_id}',  
currency = "XTR",  
prices = prices,  
subscription_period = plan["period"]  
)
```

```
return link
```

```
async def handle_subscription_payment(self, user_id, payload,  
charge_id):  
parts = payload.split(":")  
if parts[0] != "subscription":  
return False
```

```
plan_id = parts[1]  
plan = self.plans.get(plan_id)
```

```
if not plan:  
return False
```

```
await self.activate_subscription(user_id, plan_id, plan, charge_id)
```

```
return True
```

```
async def activate_subscription(self, user_id, plan_id, plan,  
charge_id):  
import time
```

```
now = int(time.time())  
expires_at = now + plan["period"]
```

```
await self.db.execute("""
```

```
INSERT INTO subscriptions (user_id, plan_id, charge_id, started_at,
expires_at, status, features)
VALUES ($1, $2, $3, $4, $5, 'active', $6)
ON CONFLICT (user_id, plan_id) DO UPDATE SET
charge_id = EXCLUDED.charge_id,
expires_at = EXCLUDED.expires_at,
status = 'active'
""", user_id, plan_id, charge_id, now, expires_at, plan["features"]])
```

```
async def check_subscription(self, user_id, feature = None):
import time
```

```
row = await self.db.fetchrow("""
SELECT * FROM subscriptions
WHERE user_id = $1 AND status = 'active' AND expires_at > $2
ORDER BY expires_at DESC
LIMIT 1
""", user_id, int(time.time()))
```

```
if not row:
return False
```

```
if feature:
return feature in row["features"]
```

```
return True
```

```
async def get_user_subscription(self, user_id):
import time
```

```
return await self.db.fetchrow("""
SELECT s.*, p.name as plan_name
FROM subscriptions s
JOIN subscription_plans p ON s.plan_id = p.id
WHERE s.user_id = $1 AND s.status = 'active' AND s.expires_at >
$2
ORDER BY s.expires_at DESC
LIMIT 1
""", user_id, int(time.time()))
```

```

async def handle_subscription_renewal(self, user_id, charge_id):
    existing = await self.get_user_subscription(user_id)

    if not existing:
        return False

    plan = self.plans.get(existing["plan_id"])

    if not plan:
        return False

    import time
    new_expires = existing["expires_at"] + plan["period"]

    await self.db.execute("""
    UPDATE subscriptions
    SET expires_at = $1, charge_id = $2
    WHERE user_id = $3 AND plan_id = $4
    """, new_expires, charge_id, user_id, existing["plan_id"])

    return True

```

Subscription Database Schema:

```

CREATE TABLE subscription_plans (
    id VARCHAR(50) PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    stars_amount INTEGER NOT NULL,
    period_seconds INTEGER NOT NULL,
    features JSONB NOT NULL DEFAULT '[]',
    is_active BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMP DEFAULT NOW()
);

CREATE TABLE subscriptions (
    id SERIAL PRIMARY KEY,
    user_id BIGINT NOT NULL,
    plan_id VARCHAR(50) REFERENCES subscription_plans(id),
    charge_id VARCHAR(100),

```

```
started_at BIGINT NOT NULL,  
expires_at BIGINT NOT NULL,  
status VARCHAR(20) DEFAULT 'active',  
features JSONB NOT NULL DEFAULT '[]',  
created_at TIMESTAMP DEFAULT NOW(),  
updated_at TIMESTAMP DEFAULT NOW(),  
UNIQUE(user_id, plan_id)  
);
```

```
CREATE INDEX idx_subscriptions_user ON subscriptions(user_id);  
CREATE INDEX idx_subscriptions_expires ON  
subscriptions(expires_at);  
CREATE INDEX idx_subscriptions_status ON subscriptions(status);
```

```
INSERT INTO subscription_plans (id, name, stars_amount,  
period_seconds, features) VALUES  
(  
    'monthly_basic', 'Basic Monthly', 100, 2592000, ['price_alerts',  
    'portfolio']),  
(  
    'monthly_pro', 'Pro Monthly', 500, 2592000, ['price_alerts',  
    'portfolio', 'trading', 'analytics']),  
(  
    'yearly_pro', 'Pro Yearly', 5000, 31536000, ['price_alerts',  
    'portfolio', 'trading', 'analytics', 'priority_support']);
```

SECTION 5: PAYMENT WEBHOOKS AND VERIFICATION

Production systems need webhook verification and idempotent payment processing.

Payment Webhook Handler:

```
from fastapi import FastAPI, Request, HTTPException  
from telegram import Update, Bot  
import hashlib  
import hmac
```

```
app = FastAPI()
```

```
class PaymentWebhookHandler:
```

```

def __init__(self, bot_token, database):
    self.bot_token = bot_token
    self.bot = Bot(bot_token)
    self.db = database

def verify_telegram_hash(self, data_check_string, received_hash):
    secret_key = hashlib.sha256(self.bot_token.encode()).digest()
    computed_hash = hmac.new(
        secret_key,
        data_check_string.encode(),
        hashlib.sha256
    ).hexdigest()

    return hmac.compare_digest(computed_hash, received_hash)

async def process_payment(self, payment_data):
    charge_id = payment_data.get("telegram_payment_charge_id")

    existing = await self.db.fetchrow(
        "SELECT id FROM processed_payments WHERE charge_id = $1",
        charge_id
    )

    if existing:
        return {"status": "already_processed", "charge_id": charge_id}

    async with self.db.transaction():
        await self.db.execute("""
            INSERT INTO processed_payments (charge_id, user_id, amount,
            payload, processed_at)
            VALUES ($1, $2, $3, $4, NOW())
            """, charge_id, payment_data["user_id"], payment_data["amount"],
            payment_data["payload"])

    await self.fulfill_purchase(payment_data)

    return {"status": "success", "charge_id": charge_id}

async def fulfill_purchase(self, payment_data):
    payload = payment_data["payload"]

```

```

user_id = payment_data["user_id"]

if payload.startswith("subscription:"):
    plan_id = payload.split(":")[1]
    await self.activate_subscription(user_id, plan_id)

elif payload.startswith("product:"):
    product_id = payload.split(":")[1]
    await self.deliver_product(user_id, product_id)

elif payload.startswith("unlock:"):
    content_id = payload.split(":")[1]
    await self.unlock_content(user_id, content_id)

elif payload.startswith("tip:"):
    creator_id = payload.split(":")[1]
    await self.process_tip(user_id, creator_id, payment_data["amount"])

async def activate_subscription(self, user_id, plan_id):
    print(f'Activating {plan_id} for user {user_id}')

async def deliver_product(self, user_id, product_id):
    print(f'Delivering {product_id} to user {user_id}')

async def unlock_content(self, user_id, content_id):
    print(f'Unlocking {content_id} for user {user_id}')

async def process_tip(self, from_user, to_user, amount):
    print(f'Tip of {amount} Stars from {from_user} to {to_user}')

```

Processed Payments Table:

```

CREATE TABLE processed_payments (
    id SERIAL PRIMARY KEY,
    charge_id VARCHAR(100) UNIQUE NOT NULL,
    user_id BIGINT NOT NULL,
    amount INTEGER NOT NULL,
    payload TEXT NOT NULL,
    status VARCHAR(20) DEFAULT 'completed',

```



```
processed_at TIMESTAMP DEFAULT NOW(),  
metadata JSONB DEFAULT '{}'  
);
```

```
CREATE INDEX idx_processed_payments_user ON  
processed_payments(user_id);  
CREATE INDEX idx_processed_payments_charge ON  
processed_payments(charge_id);
```

SECTION 6: STAR BALANCE AND WITHDRAWALS

Developers can check their Star balance and withdraw to Toncoin through Fragment.

Checking Star Balance:

```
class StarBalanceManager:
```

```
    def __init__(self, bot):  
        self.bot = bot
```

```
    async def get_star_transactions(self, offset = None, limit = 100):  
        transactions = await self.bot.get_star_transactions(  
            offset = offset,  
            limit = limit  
        )
```

```
    return transactions
```

```
    async def calculate_balance(self):  
        total_received = 0  
        total_withdrawn = 0
```

```
        offset = None
```

```
        while True:  
            transactions = await self.get_star_transactions(offset = offset)
```

```
            if not transactions.transactions:
```

break

```
for tx in transactions.transactions:
```

```
    if tx.amount > 0:
```

```
        total_received += tx.amount
```

```
    else:
```

```
        total_withdrawn += abs(tx.amount)
```

```
if len(transactions.transactions) < 100:
```

```
    break
```

```
offset = transactions.transactions[-1].id
```

```
return {
```

```
    "total_received": total_received,
```

```
    "total_withdrawn": total_withdrawn,
```

```
    "balance": total_received - total_withdrawn
```

```
}
```

```
async def get_recent_transactions(self, limit=20):
```

```
    transactions = await self.get_star_transactions(limit=limit)
```

```
    return [{
```

```
        "id": tx.id,
```

```
        "amount": tx.amount,
```

```
        "date": tx.date.isoformat() if tx.date else None,
```

```
        "source": self.identify_source(tx)
```

```
    ] for tx in transactions.transactions]
```

```
def identify_source(self, transaction):
```

```
    if hasattr(transaction, 'source') and transaction.source:
```

```
        return transaction.source.type
```

```
    return "unknown"
```

Withdrawal Process:

Withdrawals happen through Fragment (fragment.com), not the Bot API directly.

The withdrawal flow:

1. Accumulate at least 1000 Stars
2. Wait 21 days after receiving Stars (hold period)
3. Go to fragment.com and connect Telegram account
4. Connect TON wallet
5. Request withdrawal
6. Stars convert to TON at current rate
7. TON arrives in your wallet

class WithdrawalTracker:

```
def __init__(self, database):
```

```
    self.db = database
```

```
    self.min_withdrawal = 1000
```

```
    self.hold_days = 21
```

```
    async def get_withdrawable_balance(self, bot_id):
```

```
        import time
```

```
        hold_cutoff = int(time.time()) - (self.hold_days * 86400)
```

```
        row = await self.db.fetchrow("""
```

```
        SELECT COALESCE(SUM(amount), 0) as withdrawable
```

```
        FROM star_transactions
```

```
        WHERE bot_id = $1
```

```
        AND amount > 0
```

```
        AND created_at < $2
```

```
        AND NOT is_withdrawn
```

```
        """, bot_id, hold_cutoff)
```

```
        return row["withdrawable"] if row else 0
```

```
    async def get_pending_balance(self, bot_id):
```

```
        import time
```

```
        hold_cutoff = int(time.time()) - (self.hold_days * 86400)
```

```
        row = await self.db.fetchrow("""
```

```
        SELECT COALESCE(SUM(amount), 0) as pending
```

```
FROM star_transactions
WHERE bot_id = $1
AND amount > 0
AND created_at >= $2
AND NOT is_withdrawn
""", bot_id, hold_cutoff)
```

```
return row["pending"] if row else 0
```

```
async def can_withdraw(self, bot_id):
    withdrawable = await self.get_withdrawable_balance(bot_id)
    return withdrawable >= self.min_withdrawal
```

```
async def get_withdrawal_summary(self, bot_id):
    withdrawable = await self.get_withdrawable_balance(bot_id)
    pending = await self.get_pending_balance(bot_id)
```

```
    return {
        "withdrawable": withdrawable,
        "pending": pending,
        "total": withdrawable + pending,
        "min_withdrawal": self.min_withdrawal,
        "can_withdraw": withdrawable >= self.min_withdrawal,
        "hold_days": self.hold_days
    }
```

SECTION 7: REFUNDS AND DISPUTES

Handling refund requests and payment disputes.

Refund Manager:

```
class RefundManager:
```

```
    def __init__(self, bot, database):
        self.bot = bot
        self.db = database
```

```
    async def process_refund(self, user_id, charge_id, reason=None):
```

```
payment = await self.db.fetchrow("""
SELECT * FROM processed_payments
WHERE charge_id = $1 AND user_id = $2
""", charge_id, user_id)
```

```
if not payment:
```

```
    return {"success": False, "error": "Payment not found"}
```

```
if payment["status"] == "refunded":
```

```
    return {"success": False, "error": "Already refunded"}
```

```
try:
```

```
    await self.bot.refund_star_payment(
```

```
        user_id=user_id,
```

```
        telegram_payment_charge_id=charge_id
```

```
    )
```

```
await self.db.execute("""
```

```
UPDATE processed_payments
```

```
SET status = 'refunded', metadata = metadata || $1
```

```
WHERE charge_id = $2
```

```
""", {"refund_reason": reason, "refunded_at": "now"}, charge_id)
```

```
await self.revoke_purchase(payment)
```

```
return {"success": True, "refunded_amount": payment["amount"]}
```

```
except Exception as e:
```

```
    return {"success": False, "error": str(e)}
```

```
async def revoke_purchase(self, payment):
```

```
    payload = payment["payload"]
```

```
    user_id = payment["user_id"]
```

```
    if payload.startswith("subscription:"):
        await self.cancel_subscription(user_id, payload.split(":")[1])
```

```
    elif payload.startswith("product:"):
        await self.revoke_product(user_id, payload.split(":")[1])
```

```
elif payload.startswith("unlock:"):
    await self.revoke_unlock(user_id, payload.split(":")[1])
```

```
async def cancel_subscription(self, user_id, plan_id):
    await self.db.execute("""
UPDATE subscriptions
SET status = 'refunded'
WHERE user_id = $1 AND plan_id = $2
""", user_id, plan_id)
```

```
async def revoke_product(self, user_id, product_id):
    await self.db.execute("""
DELETE FROM user_products
WHERE user_id = $1 AND product_id = $2
""", user_id, product_id)
```

```
async def revoke_unlock(self, user_id, content_id):
    await self.db.execute("""
DELETE FROM content_unlocks
WHERE user_id = $1 AND content_id = $2
""", user_id, content_id)
```

Refund Command Handler:

```
class RefundBot:
```

```
    def __init__(self, token, database):
        self.token = token
        self.db = database
        self.application = Application.builder().token(token).build()
        self.refund_manager = None
```

```
    async def post_init(self, application):
        self.refund_manager = RefundManager(application.bot, self.db)
```

```
    async def cmd_refund(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
        user_id = update.effective_user.id
```

```
recent_payments = await self.db.fetch("""
SELECT charge_id, amount, payload, processed_at
FROM processed_payments
WHERE user_id = $1 AND status = 'completed'
ORDER BY processed_at DESC
LIMIT 5
""", user_id)
```

```
if not recent_payments:
    await update.message.reply_text("No recent purchases found.")
    return
```

```
keyboard = []
```

```
for payment in recent_payments:
    item_name = self.get_item_name(payment["payload"])
    button_text = f'{item_name} - {payment["amount"]} Stars'
```

```
keyboard.append([
    InlineKeyboardButton(
        button_text,
        callback_data = f'refund:{payment["charge_id"]}'
    )
])
```

```
keyboard.append([
    InlineKeyboardButton("Cancel", callback_data = "refund:cancel")
])
```

```
await update.message.reply_text(
    "Select a purchase to refund:",
    reply_markup = InlineKeyboardMarkup(keyboard)
)
```

```
async def handle_refund_callback(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
    query = update.callback_query
    await query.answer()
```

```
data = query.data.split(":")
```

```

if data[1] == "cancel":
    await query.message.edit_text("Refund cancelled.")
    return

charge_id = data[1]
user_id = update.effective_user.id

result = await self.refund_manager.process_refund(
    user_id=user_id,
    charge_id=charge_id,
    reason="User requested"
)

if result["success"]:
    await query.message.edit_text(
        f'Refund processed!\n\n'
        f'{result["refunded_amount"]} Stars have been returned to your
        account.'
    )
else:
    await query.message.edit_text(
        f'Refund failed: {result["error"]}'
    )

def get_item_name(self, payload):
    if payload.startswith("subscription:"):
        return f'Subscription ({payload.split(':')[1]})'
    elif payload.startswith("product:"):
        return f'Product ({payload.split(':')[1]})'
    elif payload.startswith("unlock:"):
        return f'Content Unlock'
    return payload

```

SECTION 8: ANALYTICS AND REPORTING

Track revenue, conversion rates, and payment patterns.

Payment Analytics:


```
class PaymentAnalytics:
```

```
    def __init__(self, database):  
        self.db = database
```

```
    async def get_daily_revenue(self, days = 30):  
        rows = await self.db.fetch("""  
        SELECT  
        DATE(processed_at) as date,  
        COUNT(*) as transactions,  
        SUM(amount) as total_stars  
        FROM processed_payments  
        WHERE processed_at >= NOW() - INTERVAL '%s days'  
        AND status = 'completed'  
        GROUP BY DATE(processed_at)  
        ORDER BY date DESC  
        """, days)
```

```
        return [dict(row) for row in rows]
```

```
    async def get_revenue_by_product(self, days = 30):  
        rows = await self.db.fetch("""  
        SELECT  
        SPLIT_PART(payload, ':', 1) as product_type,  
        COUNT(*) as transactions,  
        SUM(amount) as total_stars,  
        AVG(amount) as avg_amount  
        FROM processed_payments  
        WHERE processed_at >= NOW() - INTERVAL '%s days'  
        AND status = 'completed'  
        GROUP BY SPLIT_PART(payload, ':', 1)  
        ORDER BY total_stars DESC  
        """, days)
```

```
        return [dict(row) for row in rows]
```

```
    async def get_top_customers(self, limit = 10, days = 30):  
        rows = await self.db.fetch("""  
        SELECT
```

```
user_id,  
COUNT(*) as transactions,  
SUM(amount) as total_spent,  
MAX(processed_at) as last_purchase  
FROM processed_payments  
WHERE processed_at >= NOW() - INTERVAL '%s days'  
AND status = 'completed'  
GROUP BY user_id  
ORDER BY total_spent DESC  
LIMIT %s  
""", days, limit)
```

```
return [dict(row) for row in rows]
```

```
async def get_conversion_funnel(self, days=7):  
    invoice_views = await self.db.fetchval("""  
    SELECT COUNT(*) FROM invoice_events  
    WHERE event_type = 'view'  
    AND created_at >= NOW() - INTERVAL '%s days'  
    """, days)
```

```
    checkouts_started = await self.db.fetchval("""  
    SELECT COUNT(*) FROM invoice_events  
    WHERE event_type = 'checkout_started'  
    AND created_at >= NOW() - INTERVAL '%s days'  
    """, days)
```

```
    completed = await self.db.fetchval("""  
    SELECT COUNT(*) FROM processed_payments  
    WHERE processed_at >= NOW() - INTERVAL '%s days'  
    AND status = 'completed'  
    """, days)
```

```
    return {  
        "invoice_views": invoice_views or 0,  
        "checkouts_started": checkouts_started or 0,  
        "completed": completed or 0,  
        "view_to_checkout_rate": (checkouts_started / invoice_views * 100)  
        if invoice_views else 0,  
        "checkout_to_complete_rate": (completed / checkouts_started * 100)
```

```
if checkouts_started else 0,  
"overall_conversion": (completed / invoice_views * 100) if  
invoice_views else 0  
}
```

```
async def get_subscription_metrics(self):  
    active = await self.db.fetchval("""  
    SELECT COUNT(*) FROM subscriptions WHERE status = 'active'  
    """)
```

```
    mrr = await self.db.fetchval("""  
    SELECT COALESCE(SUM(  
    CASE  
    WHEN s.plan_id LIKE '%yearly%' THEN p.stars_amount / 12  
    ELSE p.stars_amount  
    END  
    ), 0) as mrr  
    FROM subscriptions s  
    JOIN subscription_plans p ON s.plan_id = p.id  
    WHERE s.status = 'active'  
    """)
```

```
    churn = await self.calculate_churn_rate()
```

```
    return {  
        "active_subscriptions": active,  
        "mrr_stars": mrr,  
        "churn_rate": churn  
    }
```

```
async def calculate_churn_rate(self, days=30):  
    start_count = await self.db.fetchval("""  
    SELECT COUNT(DISTINCT user_id)  
    FROM subscriptions  
    WHERE status = 'active'  
    AND started_at < EXTRACT(EPOCH FROM NOW()) - INTERVAL  
    '%s days')  
    """, days)
```

```
    churned = await self.db.fetchval("""
```

```
SELECT COUNT(DISTINCT user_id)
FROM subscriptions
WHERE status = 'cancelled'
AND updated_at >= NOW() - INTERVAL '%s days'
      "", days)
```

```
if start_count == 0:
    return 0
```

```
return (churned / start_count) * 100
```

SECTION 9: PRICING STRATEGIES

Effective pricing for digital goods and subscriptions.

Dynamic Pricing:

```
class PricingEngine:
```

```
    def __init__(self, database):
        self.db = database
```

```
        self.base_prices = {
            "premium_access": 100,
            "pro_upgrade": 500,
            "whale_tier": 1000
        }
```

```
    def get_regional_multiplier(self, language_code):
        multipliers = {
            "en": 1.0,
            "ru": 0.6,
            "de": 1.1,
            "fr": 1.0,
            "es": 0.8,
            "pt": 0.7,
            "id": 0.5,
            "vi": 0.5,
            "th": 0.6,
```

```
"tr": 0.6
```

```
}
```

```
return multipliers.get(language_code, 1.0)
```

```
def calculate_price(self, product_id, language_code=None):
```

```
    base_price = self.base_prices.get(product_id)
```

```
    if not base_price:
```

```
        return None
```

```
    if language_code:
```

```
        multiplier = self.get_regional_multiplier(language_code)
```

```
        adjusted_price = int(base_price * multiplier)
```

```
        return max(1, adjusted_price)
```

```
    return base_price
```

```
async def get_personalized_price(self, user_id, product_id):
```

```
    base_price = self.base_prices.get(product_id)
```

```
    if not base_price:
```

```
        return None
```

```
    purchase_history = await self.db.fetch("""
```

```
SELECT COUNT(*) as purchases, SUM(amount) as total_spent
```

```
FROM processed_payments
```

```
WHERE user_id = $1 AND status = 'completed'
```

```
""", user_id)
```

```
    loyalty_discount = 0
```

```
    if purchase_history:
```

```
        purchases = purchase_history[0]["purchases"] or 0
```

```
        total_spent = purchase_history[0]["total_spent"] or 0
```

```
    if total_spent > 5000:
```

```
        loyalty_discount = 0.15
```

```
    elif total_spent > 1000:
```

```
        loyalty_discount = 0.10
```

```

elif purchases > 5:
    loyalty_discount = 0.05

final_price = int(base_price * (1 - loyalty_discount))

return max(1, final_price)

def calculate_bundle_price(self, product_ids, discount_percent=20):
    total = sum(self.base_prices.get(pid, 0) for pid in product_ids)
    discounted = int(total * (1 - discount_percent / 100))
    return max(1, discounted)

```

A/B Testing Prices:

```

class PriceExperiment:

    def __init__(self, database):
        self.db = database

    async def get_experiment_price(self, user_id, experiment_id):
        bucket = user_id % 100

        experiment = await self.db.fetchrow("""
        SELECT * FROM price_experiments
        WHERE id = $1 AND is_active = TRUE
        """, experiment_id)

        if not experiment:
            return None

        variants = experiment["variants"]

        cumulative = 0
        for variant in variants:
            cumulative += variant["percent"]
            if bucket < cumulative:
                return variant["price"]

        return variants[-1]["price"]

```

```

async def record_conversion(self, user_id, experiment_id, converted,
amount = None):
    await self.db.execute("""
INSERT INTO experiment_results
(experiment_id, user_id, variant_bucket, converted, amount,
created_at)
VALUES ($1, $2, $3, $4, $5, NOW())
""", experiment_id, user_id, user_id % 100, converted, amount)

async def get_experiment_results(self, experiment_id):
    results = await self.db.fetch("""
SELECT
CASE
WHEN variant_bucket < 50 THEN 'control'
ELSE 'variant'
END as variant,
COUNT(*) as impressions,
SUM(CASE WHEN converted THEN 1 ELSE 0 END) as conversions,
SUM(amount) as revenue
FROM experiment_results
WHERE experiment_id = $1
GROUP BY CASE WHEN variant_bucket < 50 THEN 'control' ELSE
'variant' END
""", experiment_id)

    return [dict(row) for row in results]

```

SECTION 10: COMPLETE PAYMENT INTEGRATION

Full payment system combining all components.

Integrated Payment Service:

```

class PaymentService:

    def __init__(self, bot, database):
        self.bot = bot
        self.db = database

```

```
self.subscription_manager = SubscriptionManager(bot, database)
self.refund_manager = RefundManager(bot, database)
self.analytics = PaymentAnalytics(database)
self.pricing = PricingEngine(database)
```

```
async def create_payment(self, user_id, product_id,
language_code=None):
    price = self.pricing.calculate_price(product_id, language_code)
```

```
if not price:
    raise ValueError(f"Unknown product: {product_id}")
```

```
product = await self.get_product(product_id)
```

```
link = await self.bot.create_invoice_link(
    title=product["name"],
    description=product["description"],
    payload=f'product:{product_id}',
    currency="XTR",
    prices=[LabeledPrice(label=product["name"], amount=price)]
)
```

```
await self.db.execute("""
INSERT INTO invoice_events (user_id, product_id, price, event_type,
created_at)
VALUES ($1, $2, $3, 'created', NOW())
""", user_id, product_id, price)
```

```
return {
    "link": link,
    "price": price,
    "product": product
}
```

```
async def get_product(self, product_id):
    return await self.db.fetchrow("""
SELECT * FROM products WHERE id = $1
""", product_id)
```

```
async def process_successful_payment(self, user_id, payment_data):
```



```
charge_id = payment_data.telegram_payment_charge_id
payload = payment_data.invoice_payload
amount = payment_data.total_amount
```

```
existing = await self.db.fetchrow("""
SELECT id FROM processed_payments WHERE charge_id = $1
""", charge_id)
```

```
if existing:
    return {"status": "already_processed"}
```

```
async with self.db.transaction():
    await self.db.execute("""
INSERT INTO processed_payments
(charge_id, user_id, amount, payload, status, processed_at)
VALUES ($1, $2, $3, $4, 'completed', NOW())
""", charge_id, user_id, amount, payload)
```

```
await self.fulfill_purchase(user_id, payload, amount, charge_id)
```

```
return {"status": "success", "charge_id": charge_id}
```

```
async def fulfill_purchase(self, user_id, payload, amount,
charge_id):
```

```
    parts = payload.split(":")
    purchase_type = parts[0]
    purchase_id = parts[1] if len(parts) > 1 else None
```

```
    if purchase_type == "subscription":
        await self.subscription_manager.handle_subscription_payment(
            user_id, payload, charge_id
        )
```

```
    elif purchase_type == "product":
        await self.deliver_product(user_id, purchase_id)
```

```
    elif purchase_type == "unlock":
        await self.unlock_content(user_id, purchase_id)
```

```
    elif purchase_type == "tip":
```

```
await self.process_tip(user_id, purchase_id, amount)
```

```
async def deliver_product(self, user_id, product_id):  
    await self.db.execute("""  
    INSERT INTO user_products (user_id, product_id, purchased_at)  
    VALUES ($1, $2, NOW())  
    ON CONFLICT (user_id, product_id) DO NOTHING  
    """, user_id, product_id)
```

```
async def unlock_content(self, user_id, content_id):  
    await self.db.execute("""  
    INSERT INTO content_unlocks (user_id, content_id, unlocked_at)  
    VALUES ($1, $2, NOW())  
    ON CONFLICT (user_id, content_id) DO NOTHING  
    """, user_id, content_id)
```

```
async def process_tip(self, from_user, to_user, amount):  
    await self.db.execute("""  
    INSERT INTO tips (from_user_id, to_user_id, amount, created_at)  
    VALUES ($1, $2, $3, NOW())  
    """, from_user, to_user, amount)
```

```
async def check_access(self, user_id, feature):  
    has_subscription = await  
self.subscription_manager.check_subscription(  
    user_id, feature  
)
```

```
if has_subscription:  
    return True
```

```
has_product = await self.db.fetchval("""  
SELECT EXISTS(  
    SELECT 1 FROM user_products up  
    JOIN products p ON up.product_id = p.id  
    WHERE up.user_id = $1 AND $2 = ANY(p.features)  
    )  
    """, user_id, feature)
```

```
return has_product
```

```

async def get_user_purchases(self, user_id):
    purchases = await self.db.fetch("""
SELECT pp.*, p.name as product_name
FROM processed_payments pp
LEFT JOIN products p ON pp.payload = 'product:' || p.id
WHERE pp.user_id = $1 AND pp.status = 'completed'
ORDER BY pp.processed_at DESC
""", user_id)

    return [dict(row) for row in purchases]

async def get_dashboard_stats(self):
    daily = await self.analytics.get_daily_revenue(days=7)
    products = await self.analytics.get_revenue_by_product(days=30)
    subscriptions = await self.analytics.get_subscription_metrics()

    total_7_days = sum(day["total_stars"] for day in daily)
    total_30_days = sum(
        product["total_stars"] for product in products
    )

    return {
        "revenue_7_days": total_7_days,
        "revenue_30_days": total_30_days,
        "daily_breakdown": daily,
        "product_breakdown": products,
        "subscriptions": subscriptions
    }

```

CHAPTER SUMMARY

This chapter covered Telegram Stars and payments:

Telegram Stars (XTR) are mandatory for digital goods in Mini Apps. Users buy Stars via in-app purchases, spend them in bots and Mini Apps.

Creating invoices uses the Bot API. `reply_invoice` sends directly,

`create_invoice_link` generates shareable links.

Subscriptions enable recurring revenue. Set `subscription_period` for automatic renewal. Handle both initial and renewal payments.

Pre-checkout validation prevents invalid purchases. Always verify payload, amount, and user eligibility before accepting payment.

Idempotent processing prevents duplicate fulfillment. Check `charge_id` before processing. Use database transactions.

Withdrawals happen through Fragment. Minimum 1000 Stars, 21-day hold period, conversion to TON.

Refunds return Stars to users. Call `refund_star_payment`, then revoke the purchase.

Analytics track revenue and conversions. Monitor daily revenue, product performance, and subscription metrics.

Pricing strategies optimize revenue. Regional pricing, loyalty discounts, bundle pricing, A/B testing.

Next chapter: TON Blockchain Integration. We dive deep into building on TON.

CHAPTER 5: TON BLOCKCHAIN INTEGRATION

Deep integration with The Open Network. TON is the exclusive blockchain for Telegram Mini Apps since January 2025. This chapter covers TON architecture, TON Connect implementation, smart contract interaction, Jettons (tokens), NFTs, and transaction monitoring.

SECTION 1: TON ARCHITECTURE FUNDAMENTALS

TON differs significantly from Ethereum. Understanding these differences prevents building the wrong patterns.

Account Model:

In TON, everything is a smart contract. User wallets are smart contracts. There are no "externally owned accounts" like Ethereum. When you send TON to an address, you're sending a message to a contract.

Wallet contracts come in versions. v3R2 and v4R2 are common. Each version has different features and gas costs. Your code should handle multiple wallet versions.

Workchains and Shardchains:

TON uses sharding for scalability. The masterchain coordinates. Workchains handle execution. The basechain (workchain 0) is where most contracts live.

Addresses have two formats:

Raw format: 0:1234...abcd (workchain:hex)

User-friendly format: EQAbc...xyz (base64 with checksum)

Both represent the same address. User-friendly is for display, raw for computation.

Message-Based Communication:

Contracts communicate through messages, not direct calls. When you call a contract, you send a message. The contract processes it asynchronously. Responses come as separate messages.

This enables true parallelism but complicates programming:

No immediate return values from contract calls

Multiple transactions may be needed for complex operations

State changes are eventual, not instant

TON Client Setup:

```
from tonsdk.client import TonClient, TonClientException
from tonsdk.contract.wallet import Wallets, WalletVersionEnum
```

```
from tonsdk.utils import Address, bytes_to_b64str, b64str_to_bytes
from tonsdk.boc import Cell
import aiohttp
import asyncio
```

```
class TONClient:
```

```
    def __init__(self, api_key = None, testnet = False):
        self.api_key = api_key
```

```
    if testnet:
        self.base_url = "https://testnet.toncenter.com/api/v2"
    else:
        self.base_url = "https://toncenter.com/api/v2"
```

```
    self.headers = {}
    if api_key:
        self.headers["X-API-Key"] = api_key
```

```
    async def request(self, method, params = None):
        async with aiohttp.ClientSession() as session:
            url = f'{self.base_url}/{method}'
```

```
        async with session.get(url, params = params, headers = self.headers)
        as response:
            data = await response.json()
```

```
    if not data.get("ok"):
        raise Exception(f"TON API error: {data.get('error')}")
```

```
    return data.get("result")
```

```
    async def get_address_info(self, address):
        return await self.request("getAddressInformation", {
            "address": address
        })
```

```
    async def get_balance(self, address):
        info = await self.get_address_info(address)
        return int(info.get("balance", 0))
```

```
async def get_transactions(self, address, limit = 10, lt = None,
hash = None):
    params = {
        "address": address,
        "limit": limit
    }
```

```
    if lt:
        params["lt"] = lt
    if hash:
        params["hash"] = hash
```

```
    return await self.request("getTransactions", params)
```

```
async def send_boc(self, boc):
    return await self.request("sendBoc", {
        "boc": boc
    })
```

```
async def run_get_method(self, address, method, stack = None):
    params = {
        "address": address,
        "method": method
    }
```

```
    if stack:
        params["stack"] = stack
```

```
    return await self.request("runGetMethod", params)
```

```
async def estimate_fee(self, address, body, init_code = None,
init_data = None):
    params = {
        "address": address,
        "body": body
    }
```

```
    if init_code:
        params["init_code"] = init_code
```

```
if init_data:
    params["init_data"] = init_data

return await self.request("estimateFee", params)
```

SECTION 2: TON CONNECT DEEP DIVE

TON Connect is the mandatory wallet connection protocol. Understanding its internals helps debug issues and build robust integrations.

TON Connect Protocol Flow:

1. App creates connection request with manifest URL
2. User selects wallet app
3. Wallet displays connection request
4. User approves connection
5. Wallet sends proof of ownership
6. App verifies proof and establishes session
7. Session persists until disconnect

Manifest Requirements:

Your app needs a manifest file at a public URL:

```
{
  "url": "https://your-app.com",
  "name": "Crypto Trading Bot",
  "iconUrl": "https://your-app.com/icon-256.png",
  "termsOfUseUrl": "https://your-app.com/terms",
  "privacyPolicyUrl": "https://your-app.com/privacy"
}
```

The iconUrl must be 256x256 PNG. Wallets display this to users.

Complete TON Connect Implementation:

```
import TonConnect from '@tonconnect/sdk';
```



```

class TONConnectManager {

  constructor(manifestUrl) {
    this.manifestUrl = manifestUrl;
    this.connector = null;
    this.wallet = null;
    this.listeners = [];
  }

  async initialize() {
    this.connector = new TonConnect({
      manifestUrl: this.manifestUrl
    });

    this.connector.onStatusChange((wallet) => {
      this.wallet = wallet;
      this.notifyListeners(wallet);
    });

    const restoredConnection = await
    this.connector.restoreConnection();

    if (restoredConnection) {
      this.wallet = this.connector.wallet;
    }

    return this.isConnected();
  }

  isConnected() {
    return this.connector?.connected || false;
  }

  getWalletInfo() {
    if (!this.wallet) return null;

    return {
      address: this.wallet.account.address,
      chain: this.wallet.account.chain,
      publicKey: this.wallet.account.publicKey,

```

```
walletStateInit: this.wallet.account.walletStateInit,  
device: {  
  platform: this.wallet.device.platform,  
  appName: this.wallet.device.appName,  
  appVersion: this.wallet.device.appVersion  
}  
};  
}
```

```
async getAvailableWallets() {  
  const walletsList = await this.connector.getWallets();
```

```
  return walletsList.map(wallet => ({  
    name: wallet.name,  
    appName: wallet.appName,  
    imageUrl: wallet.imageUrl,  
    aboutUrl: wallet.aboutUrl,  
    platforms: wallet.platforms,  
    bridgeUrl: wallet.bridgeUrl,  
    universalLink: wallet.universalLink,  
    deepLink: wallet.deepLink,  
    jsBridgeKey: wallet.jsBridgeKey,  
    injected: wallet.injected,  
    embedded: wallet.embedded  
  }));  
}
```

```
async connect(walletInfo) {  
  if (walletInfo.jsBridgeKey) {  
    return this.connector.connect({  
      jsBridgeKey: walletInfo.jsBridgeKey  
    });  
  }
```

```
  if (walletInfo.bridgeUrl) {  
    const connectionSource = {  
      bridgeUrl: walletInfo.bridgeUrl,  
      universalLink: walletInfo.universalLink  
    };  
  }
```

```

return this.connector.connect(connectionSource);
}

throw new Error('Invalid wallet connection info');
}

async connectTonkeeper() {
const wallets = await this.getAvailableWallets();
const tonkeeper = wallets.find(w => w.appName.toLowerCase()
== 'tonkeeper');

if (!tonkeeper) {
throw new Error('Tonkeeper not available');
}

return this.connect(tonkeeper);
}

async disconnect() {
if (this.connector) {
await this.connector.disconnect();
this.wallet = null;
}
}

async sendTransaction(transaction) {
if (!this.isConnected()) {
throw new Error('Wallet not connected');
}

const result = await this.connector.sendTransaction(transaction);
return result;
}

buildTransaction(messages, validUntil = null) {
if (!validUntil) {
validUntil = Math.floor(Date.now() / 1000) + 600;
}

return {

```

```

validUntil: validUntil,
messages: messages.map(msg => ({
address: msg.address,
amount: msg.amount.toString(),
payload: msg.payload || undefined,
stateInit: msg.stateInit || undefined
}))
};
}

```

```

async sendTON(recipient, amountNano, comment = "") {
const messages = [{
address: recipient,
amount: amountNano
}];

```

```

if (comment) {
messages[0].payload = this.createCommentPayload(comment);
}

```

```

const transaction = this.buildTransaction(messages);
return this.sendTransaction(transaction);
}

```

```

createCommentPayload(text) {
const encoder = new TextEncoder();
const bytes = encoder.encode(text);

```

```

const payload = new Uint8Array(bytes.length + 4);
payload.set(bytes, 4);

```

```

return Buffer.from(payload).toString('base64');
}

```

```

onStatusChange(callback) {
this.listeners.push(callback);
return () => {
this.listeners = this.listeners.filter(l => l !== callback);
};
}

```

```

notifyListeners(wallet) {
  this.listeners.forEach(callback => {
    try {
      callback(wallet);
    } catch (error) {
      console.error('Listener error:', error);
    }
  });
}
}

```

Proof Verification (Server-Side):

```

import nacl from 'tweetnacl';
import { Address, Cell } from 'ton-core';

```

```

class TONConnectProofVerifier {

```

```

  constructor(validDomains) {
    this.validDomains = validDomains;
  }

```

```

  verifyProof(walletAddress, proof, payload) {
    const { timestamp, domain, signature, stateInit } = proof;

```

```

    const now = Math.floor(Date.now() / 1000);
    if (Math.abs(now - timestamp) > 300) {
      return { valid: false, error: 'Proof expired' };
    }

```

```

    if (!this.validDomains.includes(domain.value)) {
      return { valid: false, error: 'Invalid domain' };
    }

```

```

    const message = this.createMessage(
      walletAddress,
      timestamp,
      domain,

```

```

payload
);

const publicKey = this.extractPublicKey(stateInit);

const isValid = nacl.sign.detached.verify(
  message,
  Buffer.from(signature, 'base64'),
  publicKey
);

if (!isValid) {
  return { valid: false, error: 'Invalid signature' };
}

return { valid: true };
}

createMessage(address, timestamp, domain, payload) {
  const prefix = 'ton-proof-item-v2/';

  const addressCell = Address.parse(address);
  const workchain = addressCell.workChain;
  const hash = addressCell.hash;

  const timestampBuffer = Buffer.alloc(8);
  timestampBuffer.writeBigUInt64LE(BigInt(timestamp));

  const domainLengthBuffer = Buffer.alloc(4);
  domainLengthBuffer.writeUInt32LE(domain.lengthBytes);

  const domainBuffer = Buffer.from(domain.value, 'utf8');

  const payloadBuffer = Buffer.from(payload, 'utf8');

  return Buffer.concat([
    Buffer.from(prefix),
    Buffer.from([workchain]),
    hash,
    domainLengthBuffer,

```

```

domainBuffer,
timestampBuffer,
payloadBuffer
]);
}

```

```

extractPublicKey(stateInit) {
const cell = Cell.fromBase64(stateInit);
return cell.bits.buffer.slice(0, 32);
}
}

```

SECTION 3: JETTON (TOKEN) OPERATIONS

Jettons are TON's fungible tokens, similar to ERC-20. They follow a master-wallet architecture where each user has a separate wallet contract for each token.

Jetton Architecture:

Jetton Master: The main token contract. Stores total supply and metadata.

Jetton Wallet: Per-user wallet for the token. Each holder has a separate contract.

To transfer Jettons, you send a message to YOUR Jetton wallet, not the recipient's.

Jetton Client:

```

class JettonClient {

```

```

  constructor(tonClient) {
    this.ton = tonClient;
  }

```

```

  async getJettonData(masterAddress) {
    const result = await this.ton.run_get_method(
      masterAddress,

```

```
"get_jetton_data"
```

```
);
```

```
if (!result.stack || result.stack.length < 5) {  
  throw new Error("Invalid jetton data response");  
}
```

```
return {  
  totalSupply: BigInt(result.stack[0][1]),  
  mintable: result.stack[1][1] !== "0",  
  adminAddress: this.parseAddress(result.stack[2]),  
  content: this.parseContent(result.stack[3]),  
  walletCode: result.stack[4][1]  
};  
}
```

```
async getJettonWalletAddress(masterAddress, ownerAddress) {  
  const result = await this.ton.run_get_method(  
    masterAddress,  
    "get_wallet_address",  
    [["tvm.Slice", this.addressToSlice(ownerAddress)]]  
  );
```

```
if (!result.stack || result.stack.length < 1) {  
  throw new Error("Failed to get wallet address");  
}
```

```
return this.parseAddress(result.stack[0]);  
}
```

```
async getJettonWalletData(walletAddress) {  
  const result = await this.ton.run_get_method(  
    walletAddress,  
    "get_wallet_data"  
  );
```

```
if (!result.stack || result.stack.length < 4) {  
  throw new Error("Invalid wallet data response");  
}
```



```
return {  
  balance: BigInt(result.stack[0][1]),  
  ownerAddress: this.parseAddress(result.stack[1]),  
  masterAddress: this.parseAddress(result.stack[2]),  
  walletCode: result.stack[3][1]  
};  
}
```

```
async getJettonBalance(masterAddress, ownerAddress) {  
  const walletAddress = await this.getJettonWalletAddress(  
    masterAddress,  
    ownerAddress  
  );
```

```
  try {  
    const walletData = await this.getJettonWalletData(walletAddress);  
    return walletData.balance;  
  } catch (error) {  
    return BigInt(0);  
  }  
}
```

```
buildTransferPayload(destination, amount, responseAddress,  
forwardAmount, forwardPayload) {  
  const payload = {  
    op: 0xf8a7ea5,  
    queryId: Date.now(),  
    amount: amount,  
    destination: destination,  
    responseAddress: responseAddress,  
    customPayload: null,  
    forwardAmount: forwardAmount || 0,  
    forwardPayload: forwardPayload || null  
  };  
};
```

```
return this.serializeTransferPayload(payload);  
}
```

```
serializeTransferPayload(payload) {  
  return Buffer.from(JSON.stringify(payload)).toString('base64');
```

```
}
```

```
parseAddress(stackItem) {  
  if (stackItem[0] === "tvm.Cell" || stackItem[0] ===  
"tvm.Slice") {  
    return stackItem[1];  
  }  
  return stackItem[1];  
}
```

```
parseContent(stackItem) {  
  return stackItem[1];  
}
```

```
addressToSlice(address) {  
  return address;  
}  
}
```

Jetton Transfer via TON Connect:

```
class JettonTransferService {  
  
  constructor(tonConnect, jettonClient) {  
    this.connect = tonConnect;  
    this.jetton = jettonClient;  
  }  
  
  async transfer(jettonMaster, recipient, amount, comment = ") {  
    const wallet = this.connect.getWalletInfo();  
  
    if (!wallet) {  
      throw new Error('Wallet not connected');  
    }  
  
    const senderJettonWallet = await  
this.jetton.getJettonWalletAddress(  
  jettonMaster,  
  wallet.address
```

```

);

const transferPayload = this.buildTransferBody(
  recipient,
  amount,
  wallet.address,
  comment
);

const transaction = {
  validUntil: Math.floor(Date.now() / 1000) + 600,
  messages: [{
    address: senderJettonWallet,
    amount: "1000000000",
    payload: transferPayload
  }]
};

return this.connect.sendTransaction(transaction);
}

buildTransferBody(destination, jettonAmount, responseAddress,
comment) {
  return Buffer.from(JSON.stringify({
    op: "transfer",
    destination: destination,
    amount: jettonAmount.toString(),
    response_address: responseAddress,
    forward_amount: "1",
    forward_payload: comment
  })).toString('base64');
}

async getBalance(jettonMaster, ownerAddress) {
  return this.jetton.getJettonBalance(jettonMaster, ownerAddress);
}
}

```

SECTION 4: NFT OPERATIONS

TON NFTs follow a collection-item architecture. Collections contain items. Each item is a separate contract.

NFT Client:

```
class NFTClient {

  constructor(tonClient) {
    this.ton = tonClient;
  }

  async getCollectionData(collectionAddress) {
    const result = await this.ton.run_get_method(
      collectionAddress,
      "get_collection_data"
    );

    return {
      nextItemIndex: BigInt(result.stack[0][1]),
      content: this.parseContent(result.stack[1]),
      ownerAddress: this.parseAddress(result.stack[2])
    };
  }

  async getNFTAddressByIndex(collectionAddress, index) {
    const result = await this.ton.run_get_method(
      collectionAddress,
      "get_nft_address_by_index",
      [["num", index.toString()]]
    );

    return this.parseAddress(result.stack[0]);
  }

  async getNFTData(nftAddress) {
    const result = await this.ton.run_get_method(
      nftAddress,
      "get_nft_data"
    );
```

```

return {
  initialized: result.stack[0][1] !== "0",
  index: BigInt(result.stack[1][1]),
  collectionAddress: this.parseAddress(result.stack[2]),
  ownerAddress: this.parseAddress(result.stack[3]),
  content: this.parseContent(result.stack[4])
};
}

```

```

async getNFTOwner(nftAddress) {
  const data = await this.getNFTData(nftAddress);
  return data.ownerAddress;
}

```

```

async isOwner(nftAddress, walletAddress) {
  const owner = await this.getNFTOwner(nftAddress);
  return owner.toLowerCase() === walletAddress.toLowerCase();
}

```

```

async getOwnedNFTs(collectionAddress, ownerAddress, maxIndex
= 1000) {
  const ownedNFTs = [];
  const collectionData = await
this.getCollectionData(collectionAddress);
  const totalItems = Number(collectionData.nextItemIndex);

```

```

  const checkLimit = Math.min(totalItems, maxIndex);

```

```

  for (let i = 0; i < checkLimit; i++) {
    try {
      const nftAddress = await this.getNFTAddressByIndex(
collectionAddress,
i
);

```

```

  const nftData = await this.getNFTData(nftAddress);

```

```

  if (nftData.ownerAddress.toLowerCase() ===
ownerAddress.toLowerCase()) {

```

```

ownedNFTs.push({
  address: nftAddress,
  index: i,
  ...nftData
});
}
} catch (error) {
  continue;
}
}

return ownedNFTs;
}

parseContent(stackItem) {
  return stackItem[1];
}

parseAddress(stackItem) {
  return stackItem[1];
}
}

```

NFT Transfer via TON Connect:

```

class NFTTransferService {

  constructor(tonConnect, nftClient) {
    this.connect = tonConnect;
    this.nft = nftClient;
  }

  async transfer(nftAddress, newOwner, forwardAmount =
"50000000") {
    const wallet = this.connect.getWalletInfo();

    if (!wallet) {
      throw new Error('Wallet not connected');
    }
  }
}

```

```
const isOwner = await this.nft.isOwner(nftAddress,  
wallet.address);
```

```
if (!isOwner) {  
  throw new Error('You do not own this NFT');  
}
```

```
const transferPayload = this.buildTransferBody(  
  newOwner,  
  wallet.address,  
  forwardAmount  
);
```

```
const transaction = {  
  validUntil: Math.floor(Date.now() / 1000) + 600,  
  messages: [{  
    address: nftAddress,  
    amount: "1000000000",  
    payload: transferPayload  
  }]  
};
```

```
return this.connect.sendTransaction(transaction);  
}
```

```
buildTransferBody(newOwner, responseAddress, forwardAmount) {  
  return Buffer.from(JSON.stringify({  
    op: "transfer",  
    new_owner: newOwner,  
    response_destination: responseAddress,  
    forward_amount: forwardAmount  
  })).toString('base64');  
}
```

```
async verifyOwnership(collectionAddress, walletAddress, minCount  
= 1) {  
  const ownedNFTs = await this.nft.getOwnedNFTs(  
    collectionAddress,  
    walletAddress,
```

100

);

return ownedNFTs.length >= minCount;

}

}

SECTION 5: TRANSACTION MONITORING

Monitor blockchain transactions for deposits, confirmations, and events.

Transaction Monitor:

```
class TransactionMonitor {
```

```
  constructor(tonClient, database) {
```

```
    this.ton = tonClient;
```

```
    this.db = database;
```

```
    this.watchedAddresses = new Map();
```

```
    this.running = false;
```

```
    this.pollInterval = 5000;
```

```
  }
```

```
  addWatch(address, callback, options = {}) {
```

```
    this.watchedAddresses.set(address, {
```

```
      callback: callback,
```

```
      lastLt: options.lastLt || "0",
```

```
      minAmount: options.minAmount || 0
```

```
    });
```

```
  }
```

```
  removeWatch(address) {
```

```
    this.watchedAddresses.delete(address);
```

```
  }
```

```
  async start() {
```

```
    this.running = true;
```

```
    this.poll();
```



```
}
```

```
stop() {  
  this.running = false;  
}
```

```
async poll() {  
  while (this.running) {  
    for (const [address, config] of this.watchedAddresses) {  
      try {  
        await this.checkAddress(address, config);  
      } catch (error) {  
        console.error(`Error checking ${address}:`, error);  
      }  
    }  
  }  
}
```

```
await this.sleep(this.pollInterval);  
}  
}
```

```
async checkAddress(address, config) {  
  const transactions = await this.ton.get_transactions(address, 20);
```

```
  const newTransactions = transactions.filter(tx => {  
    return BigInt(tx.transaction_id.lt) > BigInt(config.lastLt);  
  });
```

```
  if (newTransactions.length === 0) return;
```

```
  newTransactions.sort((a, b) => {  
    return BigInt(a.transaction_id.lt) - BigInt(b.transaction_id.lt);  
  });
```

```
  for (const tx of newTransactions) {  
    if (this.isIncomingTransaction(tx, address)) {  
      const amount = this.extractAmount(tx);
```

```
      if (amount >= config.minAmount) {  
        await config.callback({  
          type: 'deposit',
```

```
address: address,  
amount: amount,  
from: this.extractSender(tx),  
lt: tx.transaction_id.lt,  
hash: tx.transaction_id.hash,  
timestamp: tx.utime,  
comment: this.extractComment(tx)  
});  
}  
}
```

```
config.lastLt = tx.transaction_id.lt;  
}
```

```
this.watchedAddresses.set(address, config);  
}
```

```
isIncomingTransaction(tx, address) {  
if (!tx.in_msg) return false;
```

```
const destination = tx.in_msg.destination;  
return destination && destination.toLowerCase() ===  
address.toLowerCase();  
}
```

```
extractAmount(tx) {  
if (tx.in_msg && tx.in_msg.value) {  
return BigInt(tx.in_msg.value);  
}  
return BigInt(0);  
}
```

```
extractSender(tx) {  
if (tx.in_msg && tx.in_msg.source) {  
return tx.in_msg.source;  
}  
return null;  
}
```

```
extractComment(tx) {
```

```

if (tx.in_msg && tx.in_msg.msg_data) {
  try {
    const data = tx.in_msg.msg_data;
    if (data.text) {
      return data.text;
    }
  } catch (error) {
  }
}
return null;
}

sleep(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}
}

```

Deposit Handler:

```

class DepositHandler {

  constructor(transactionMonitor, database, notificationService) {
    this.monitor = transactionMonitor;
    this.db = database;
    this.notify = notificationService;
  }

  async watchUserDeposits(userId, depositAddress) {
    await this.db.execute(
      INSERT INTO deposit_addresses (user_id, address, created_at)
      VALUES ($1, $2, NOW())
      ON CONFLICT (user_id) DO UPDATE SET address = $2
      `, userId, depositAddress);

    this.monitor.addWatch(depositAddress, async (tx) => {
      await this.handleDeposit(userId, tx);
    }, {
      minAmount: 10000000
    });
  }
}

```

```
}
```

```
async handleDeposit(userId, transaction) {  
  const existing = await this.db.fetchrow(  
    SELECT id FROM deposits  
    WHERE tx_hash = $1  
    `, transaction.hash);
```

```
  if (existing) return;
```

```
  const amountTON = Number(transaction.amount) / 1e9;
```

```
  await this.db.execute(  
    INSERT INTO deposits  
    (user_id, address, amount_nano, amount_ton, tx_hash, tx_lt, sender,  
    comment, status, created_at)  
    VALUES ($1, $2, $3, $4, $5, $6, $7, $8, 'pending', NOW())  
    `, userId, transaction.address, transaction.amount.toString(),  
    amountTON, transaction.hash, transaction.lt,  
    transaction.from, transaction.comment);
```

```
  const confirmations = await this.waitForConfirmations(  
    transaction.hash,  
    5  
  );
```

```
  if (confirmations >= 5) {  
    await this.confirmDeposit(userId, transaction.hash, amountTON);  
  }  
}
```

```
async waitForConfirmations(txHash, required, timeout = 60000) {  
  const startTime = Date.now();
```

```
  while (Date.now() - startTime < timeout) {  
    const confirmations = await this.getConfirmations(txHash);
```

```
    if (confirmations >= required) {  
      return confirmations;  
    }  
  }
```

```

await new Promise(resolve => setTimeout(resolve, 3000));
}

return 0;
}

async getConfirmations(txHash) {
return 10;
}

async confirmDeposit(userId, txHash, amount) {
await this.db.execute(
UPDATE deposits SET status = 'confirmed' WHERE tx_hash = $1
`, txHash);

await this.db.execute(
UPDATE user_balances
SET balance = balance + $1, updated_at = NOW()
WHERE user_id = $2
`, amount, userId);

await this.notify.sendDepositConfirmation(userId, amount,
txHash);
}
}

```

SECTION 6: WALLET GENERATION AND MANAGEMENT

Generate TON wallets for users. Important security considerations apply.

Wallet Generator:

```

from tonsdk.contract.wallet import Wallets, WalletVersionEnum
from tonsdk.utils import bytes_to_b64str
import hashlib
import os

```

class WalletGenerator:

```
def __init__(self, wallet_version = WalletVersionEnum.v4r2):  
    self.wallet_version = wallet_version
```

```
def generate_wallet(self):  
    mnemonics, public_key, private_key, wallet = Wallets.create(  
        version=self.wallet_version,  
        workchain=0  
    )
```

```
    address = wallet.address.to_string(True, True, True)
```

```
    return {  
        "address": address,  
        "public_key": bytes_to_b64str(public_key),  
        "private_key": bytes_to_b64str(private_key),  
        "mnemonic": mnemonics  
    }
```

```
def wallet_from_mnemonic(self, mnemonic):  
    mnemonics, public_key, private_key, wallet =  
    Wallets.from_mnemonics(  
        mnemonics=mnemonic,  
        version=self.wallet_version,  
        workchain=0  
    )
```

```
    address = wallet.address.to_string(True, True, True)
```

```
    return {  
        "address": address,  
        "public_key": bytes_to_b64str(public_key),  
        "private_key": bytes_to_b64str(private_key),  
        "wallet": wallet  
    }
```

```
def derive_deposit_address(self, master_seed, user_id):  
    derived_seed = hashlib.sha256(  
        f'{master_seed}:{user_id}'.encode()
```

```
).digest()
```

```
from tonsdk.crypto import mnemonic_from_entropy
```

```
mnemonic = mnemonic_from_entropy(derived_seed[:32])
```

```
return self.wallet_from_mnemonic(mnemonic)
```

Secure Key Storage:

```
import os
```

```
from cryptography.fernet import Fernet
```

```
from cryptography.hazmat.primitives import hashes
```

```
from cryptography.hazmat.primitives.kdf.pbkdf2 import
```

```
PBKDF2HMAC
```

```
import base64
```

```
class SecureKeyStore:
```

```
def __init__(self, master_password, salt=None):
```

```
self.salt = salt or os.urandom(16)
```

```
self.cipher = self._create_cipher(master_password)
```

```
def _create_cipher(self, password):
```

```
kdf = PBKDF2HMAC(
```

```
algorithm=hashes.SHA256(),
```

```
length=32,
```

```
salt=self.salt,
```

```
iterations=480000
```

```
)
```

```
key = base64.urlsafe_b64encode(kdf.derive(password.encode()))
```

```
return Fernet(key)
```

```
def encrypt_key(self, private_key):
```

```
if isinstance(private_key, str):
```

```
private_key = private_key.encode()
```

```
return self.cipher.encrypt(private_key).decode()
```

```

def decrypt_key(self, encrypted_key):
    if isinstance(encrypted_key, str):
        encrypted_key = encrypted_key.encode()

    return self.cipher.decrypt(encrypted_key).decode()

def store_wallet(self, database, user_id, wallet_data):
    encrypted_mnemonic = self.encrypt_key("
.join(wallet_data["mnemonic"]))
    encrypted_private_key =
self.encrypt_key(wallet_data["private_key"])

    database.execute("""
INSERT INTO user_wallets
(user_id, address, public_key, encrypted_mnemonic,
encrypted_private_key, created_at)
VALUES ($1, $2, $3, $4, $5, NOW())
""", user_id, wallet_data["address"], wallet_data["public_key"],
encrypted_mnemonic, encrypted_private_key)

def retrieve_wallet(self, database, user_id):
    row = database.fetchrow("""
SELECT * FROM user_wallets WHERE user_id = $1
""", user_id)

    if not row:
        return None

    return {
        "address": row["address"],
        "public_key": row["public_key"],
        "mnemonic": self.decrypt_key(row["encrypted_mnemonic"]).split("
"),
        "private_key": self.decrypt_key(row["encrypted_private_key"])
    }

```

SECTION 7: DEX INTEGRATION

Integrate with TON DEXs for token swaps.

DEX Client (STON.fi Example):

```
class STONfiClient {

  constructor(tonConnect) {
    this.connect = tonConnect;
    this.routerAddress =
"EQB3ncyBUTjZUA5EnFKR5_EnOMI9V1tTEAAPaiU71gc4TiUt";
  }

  async getPoolAddress(token0, token1) {
    return "EQpool_address";
  }

  async getQuote(tokenIn, tokenOut, amountIn) {
    return {
      amountIn: amountIn,
      amountOut: amountIn * 0.99,
      priceImpact: 0.01,
      fee: amountIn * 0.003
    };
  }

  async swap(tokenIn, tokenOut, amountIn, minAmountOut,
deadline = null) {
    const wallet = this.connect.getWalletInfo();

    if (!wallet) {
      throw new Error("Wallet not connected");
    }

    if (!deadline) {
      deadline = Math.floor(Date.now() / 1000) + 600;
    }

    const swapPayload = this.buildSwapPayload(
      tokenIn,
      tokenOut,
```

```
amountIn,  
minAmountOut,  
wallet.address,  
deadline  
);
```

```
const transaction = {  
  validUntil: deadline,  
  messages: [{  
    address: this.routerAddress,  
    amount: tokenIn === 'TON' ? amountIn.toString() :  
    "1000000000",  
    payload: swapPayload  
  }]  
};
```

```
return this.connect.sendTransaction(transaction);  
}
```

```
buildSwapPayload(tokenIn, tokenOut, amountIn, minAmountOut,  
recipient, deadline) {  
  return Buffer.from(JSON.stringify({  
    op: "swap",  
    token_in: tokenIn,  
    token_out: tokenOut,  
    amount_in: amountIn.toString(),  
    min_amount_out: minAmountOut.toString(),  
    recipient: recipient,  
    deadline: deadline  
  })).toString('base64');  
}
```

```
calculateMinAmountOut(expectedAmount, slippagePercent) {  
  const slippage = slippagePercent / 100;  
  return BigInt(Math.floor(Number(expectedAmount) * (1 -  
slippage)));  
}  
}
```

Swap Service:

```
class SwapService {

  constructor(dexClient, priceService) {
    this.dex = dexClient;
    this.prices = priceService;
    this.defaultSlippage = 0.5;
  }

  async getSwapQuote(tokenIn, tokenOut, amountIn) {
    const quote = await this.dex.getQuote(tokenIn, tokenOut,
amountIn);

    const tokenInPrice = await this.prices.getPrice(tokenIn);
    const tokenOutPrice = await this.prices.getPrice(tokenOut);

    return {
      ...quote,
      tokenInPrice: tokenInPrice,
      tokenOutPrice: tokenOutPrice,
      valueIn: amountIn * tokenInPrice,
      valueOut: quote.amountOut * tokenOutPrice,
      rate: quote.amountOut / amountIn
    };
  }

  async executeSwap(tokenIn, tokenOut, amountIn, slippagePercent
= null) {
    const slippage = slippagePercent || this.defaultSlippage;

    const quote = await this.getSwapQuote(tokenIn, tokenOut,
amountIn);

    const minAmountOut = this.dex.calculateMinAmountOut(
quote.amountOut,
slippage
);

    return this.dex.swap(tokenIn, tokenOut, amountIn,
```

```
minAmountOut);
}
}
```

SECTION 8: SMART CONTRACT INTERACTION

Call arbitrary smart contract methods and build custom contract integrations.

Contract Caller:

```
class ContractCaller {

  constructor(tonClient, tonConnect) {
    this.ton = tonClient;
    this.connect = tonConnect;
  }

  async callGetter(contractAddress, method, args = []) {
    const stack = args.map(arg => {
      if (typeof arg === 'bigint' || typeof arg === 'number') {
        return ["num", arg.toString()];
      }
      if (typeof arg === 'string' && arg.startsWith('EQ')) {
        return ["tvm.Slice", arg];
      }
      return ["tvm.Cell", arg];
    });

    const result = await this.ton.run_get_method(
      contractAddress,
      method,
      stack.length > 0 ? stack : undefined
    );

    return this.parseResult(result);
  }

  parseResult(result) {
```

```
if (!result.stack) return null;
```

```
return result.stack.map(item => {  
  const type = item[0];  
  const value = item[1];
```

```
  if (type === "num") {  
    return BigInt(value);  
  }
```

```
  if (type === "cell" || type === "tvm.Cell") {  
    return { type: "cell", data: value };  
  }
```

```
  if (type === "slice" || type === "tvm.Slice") {  
    return { type: "slice", data: value };  
  }
```

```
  return value;  
});  
}
```

```
async sendMessage(contractAddress, amount, payload, stateInit =  
null) {
```

```
  if (!this.connect.isConnected()) {  
    throw new Error('Wallet not connected');  
  }
```

```
  const message = {  
    address: contractAddress,  
    amount: amount.toString(),  
    payload: payload  
  };  
};
```

```
if (stateInit) {  
  message.stateInit = stateInit;  
}
```

```
const transaction = {  
  validUntil: Math.floor(Date.now() / 1000) + 600,  
  messages: [message]  
};
```

```

return this.connect.sendTransaction(transaction);
}

buildPayload(op, queryId, data = {}) {
return Buffer.from(JSON.stringify({
op: op,
query_id: queryId || Date.now(),
...data
})).toString('base64');
}
}

```

Custom Contract Integration Example:

```

class StakingContract {

  constructor(contractCaller, contractAddress) {
    this.caller = contractCaller;
    this.address = contractAddress;
  }

  async getStakingInfo() {
    const result = await this.caller.callGetter(
      this.address,
      "get_staking_info"
    );

    return {
      totalStaked: result[0],
      rewardRate: result[1],
      minStake: result[2],
      lockPeriod: result[3]
    };
  }

  async getUserStake(userAddress) {
    const result = await this.caller.callGetter(
      this.address,
      "get_user_stake",

```

```
[userAddress]
```

```
);
```

```
return {  
  stakedAmount: result[0],  
  pendingRewards: result[1],  
  stakeTime: result[2],  
  unlockTime: result[3]  
};  
}
```

```
async stake(amount) {  
  const payload = this.caller.buildPayload(  
    0x1,  
    Date.now(),  
    { action: "stake" }  
  );  
}
```

```
return this.caller.sendMessage(this.address, amount, payload);  
}
```

```
async unstake(amount) {  
  const payload = this.caller.buildPayload(  
    0x2,  
    Date.now(),  
    { action: "unstake", amount: amount.toString() }  
  );  
}
```

```
return this.caller.sendMessage(this.address, "100000000", payload);  
}
```

```
async claimRewards() {  
  const payload = this.caller.buildPayload(  
    0x3,  
    Date.now(),  
    { action: "claim" }  
  );  
}
```

```
return this.caller.sendMessage(this.address, "100000000", payload);  
}
```

```
}
```

SECTION 9: GAS AND FEE ESTIMATION

Estimate transaction fees before sending.

Fee Estimator:

```
class FeeEstimator {

  constructor(tonClient) {
    this.ton = tonClient;

    this.baseFees = {
      simpleTransfer: 100000000,
      jettonTransfer: 500000000,
      nftTransfer: 500000000,
      contractCall: 300000000
    };
  }

  async estimateTransferFee(fromAddress, toAddress, amount) {
    const mockBody = "te6ccckEBAQEAAgAAAEysuc0=";

    try {
      const result = await this.ton.estimate_fee(
        fromAddress,
        mockBody
      );

      return this.parseFeeResult(result);
    } catch (error) {
      return {
        gasFee: this.baseFees.simpleTransfer,
        storageFee: 0,
        forwardFee: 0,
        total: this.baseFees.simpleTransfer
      };
    }
  }
}
```



```
}
```

```
async estimateJettonTransferFee(senderWallet, jettonWallet) {  
  const mockBody = "te6ccckEBAQEAAgAAAEysuc0=";
```

```
  try {  
    const result = await this.ton.estimate_fee(  
      jettonWallet,  
      mockBody  
    );
```

```
    return this.parseFeeResult(result);  
  } catch (error) {  
    return {  
      gasFee: this.baseFees.jettonTransfer,  
      storageFee: 0,  
      forwardFee: 100000000,  
      total: this.baseFees.jettonTransfer + 100000000  
    };  
  }  
}
```

```
parseFeeResult(result) {  
  const fees = result.source_fees || {};
```

```
  return {  
    gasFee: parseInt(fees.gas_fee || 0),  
    storageFee: parseInt(fees.storage_fee || 0),  
    forwardFee: parseInt(fees.fwd_fee || 0),  
    total: parseInt(fees.gas_fee || 0) +  
      parseInt(fees.storage_fee || 0) +  
      parseInt(fees.fwd_fee || 0)  
  };  
}
```

```
addSafetyMargin(estimatedFee, marginPercent = 20) {  
  const margin = estimatedFee * marginPercent / 100;  
  return BigInt(Math.ceil(estimatedFee + margin));  
}
```

```

formatFee(nanoTON) {
  return (Number(nanoTON) / 1e9).toFixed(4) + " TON";
}
}

```

SECTION 10: COMPLETE TON INTEGRATION EXAMPLE

Full integration combining all components.

```

class TONIntegration {

  constructor(config) {
    this.config = config;
    this.tonClient = new TONClient(config.apiKey, config.testnet);
    this.tonConnect = null;
    this.jettonClient = null;
    this.nftClient = null;
    this.feeEstimator = null;
  }

  async initialize(manifestUrl) {
    this.tonConnect = new TONConnectManager(manifestUrl);
    await this.tonConnect.initialize();

    this.jettonClient = new JettonClient(this.tonClient);
    this.nftClient = new NFTClient(this.tonClient);
    this.feeEstimator = new FeeEstimator(this.tonClient);

    return this.tonConnect.isConnected();
  }

  isWalletConnected() {
    return this.tonConnect?.isConnected() || false;
  }

  getWalletAddress() {
    const info = this.tonConnect?.getWalletInfo();
    return info?.address;
  }
}

```

```
async connectWallet(walletName = 'Tonkeeper') {  
  return this.tonConnect.connectTonkeeper();  
}
```

```
async disconnectWallet() {  
  return this.tonConnect.disconnect();  
}
```

```
async getBalance() {  
  const address = this.getWalletAddress();  
  if (!address) return BigInt(0);  
  
  const balance = await this.tonClient.get_balance(address);  
  return BigInt(balance);  
}
```

```
async getJettonBalance(jettonMaster) {  
  const address = this.getWalletAddress();  
  if (!address) return BigInt(0);  
  
  return this.jettonClient.getJettonBalance(jettonMaster, address);  
}
```

```
async sendTON(recipient, amountNano, comment = "") {  
  return this.tonConnect.sendTON(recipient, amountNano,  
  comment);  
}
```

```
async sendJetton(jettonMaster, recipient, amount) {  
  const transferService = new JettonTransferService(  
    this.tonConnect,  
    this.jettonClient  
  );
```

```
  return transferService.transfer(jettonMaster, recipient, amount);  
}
```

```
async transferNFT(nftAddress, newOwner) {  
  const transferService = new NFTTransferService(  
    this.tonConnect,  
    this.jettonClient,  
    this.nftClient
```

```
this.tonConnect,  
this.nftClient  
);
```

```
return transferService.transfer(nftAddress, newOwner);  
}
```

```
async verifyNFTOwnership(collectionAddress, minCount = 1) {  
  const address = this.getWalletAddress();  
  if (!address) return false;
```

```
  const transferService = new NFTTransferService(  
    this.tonConnect,  
    this.nftClient  
  );
```

```
  return transferService.verifyOwnership(collectionAddress, address,  
    minCount);  
}
```

```
async getTransactionHistory(limit = 20) {  
  const address = this.getWalletAddress();  
  if (!address) return [];
```

```
  return this.tonClient.get_transactions(address, limit);  
}
```

```
async estimateFee(transactionType) {  
  const address = this.getWalletAddress();  
  if (!address) return null;
```

```
  switch (transactionType) {  
    case 'transfer':  
      return this.feeEstimator.estimateTransferFee(address, address, "0");  
    case 'jetton':  
      return { total: this.feeEstimator.baseFees.jettonTransfer };  
    case 'nft':  
      return { total: this.feeEstimator.baseFees.nftTransfer };  
    default:  
      return { total: this.feeEstimator.baseFees.simpleTransfer };
```

```
}  
}
```

```
onWalletChange(callback) {  
  return this.tonConnect.onStatusChange(callback);  
}  
}
```

CHAPTER SUMMARY

This chapter covered TON blockchain integration:

TON architecture differs from Ethereum. Everything is a contract. Communication is message-based. Sharding enables scalability.

TON Connect is mandatory for wallet integration. Manifest files identify your app. Proof verification ensures authenticity.

Jettons use master-wallet architecture. Transfer messages go to your Jetton wallet, not the recipient's.

NFTs follow collection-item structure. Each item is a separate contract. Ownership verification checks the item contract.

Transaction monitoring tracks deposits and confirmations. Poll for new transactions. Verify confirmations before crediting.

Wallet generation requires secure key storage. Encrypt private keys. Never store mnemonics in plaintext.

DEX integration enables token swaps. Build swap payloads. Handle slippage and deadlines.

Smart contract interaction uses getters and messages. Call getters for read operations. Send messages for writes.

Fee estimation prevents failed transactions. Add safety margins. Display fees to users.

Next chapter: Trading Bot Architecture. We build systems handling billions in volume.

CHAPTER 6: TRADING BOT ARCHITECTURE

How billion-dollar trading bots work. Trojan, Maestro, and Banana Gun process over 50 billion dollars annually. This chapter breaks down their architecture - token sniping, copy trading, MEV protection, risk management, and position sizing. You'll understand how to build systems that handle real money at scale.

SECTION 1: TRADING BOT ANATOMY

Trading bots are complex systems with multiple interconnected components. Each piece handles specific responsibilities.

Understanding the architecture helps you build reliable systems.

Core Components:

Telegram Interface Layer: Receives commands, sends notifications, handles user interaction. This is where users experience your bot.

Order Management System: Tracks pending orders, manages execution queue, handles retries and failures.

Execution Engine: Submits transactions to blockchain, optimizes gas, handles MEV protection.

Market Data Service: Fetches prices, monitors liquidity, tracks token information.

Risk Management: Validates positions, enforces limits, protects users from catastrophic losses.

User Database: Stores wallets, settings, positions, history. Critical for persistence.

Monitoring System: Tracks bot health, alerts on issues, logs everything.

High-Level Architecture:

class TradingBotCore:

```
def __init__(self, config):  
    self.config = config
```

```
self.telegram = TelegramInterface(config.bot_token)  
self.orders = OrderManager()  
self.executor = ExecutionEngine(config.rpc_urls)  
self.market = MarketDataService(config.price_sources)  
self.risk = RiskManager(config.risk_limits)  
self.db = Database(config.database_url)  
self.monitor = MonitoringService()
```

```
self.running = False
```

```
async def start(self):  
    self.running = True
```

```
await self.db.connect()  
await self.market.start()  
await self.executor.start()  
await self.monitor.start()
```

```
asyncio.create_task(self.order_processing_loop())  
asyncio.create_task(self.position_monitoring_loop())
```

```
await self.telegram.start()
```

```
async def stop(self):  
    self.running = False
```

```
await self.telegram.stop()  
await self.executor.stop()  
await self.market.stop()  
await self.db.disconnect()
```

```
async def order_processing_loop(self):  
    while self.running:
```

```
try:  
    pending_orders = await self.orders.get_pending()
```

```
    for order in pending_orders:  
        await self.process_order(order)
```

```
except Exception as e:  
    self.monitor.log_error("order_processing", e)
```

```
    await asyncio.sleep(0.1)
```

```
    async def process_order(self, order):  
        validation = await self.risk.validate_order(order)
```

```
        if not validation.approved:  
            await self.orders.reject(order.id, validation.reason)  
            await self.telegram.notify_order_rejected(order.user_id, order,  
validation.reason)  
        return
```

```
    try:  
        result = await self.executor.execute(order)
```

```
        if result.success:  
            await self.orders.complete(order.id, result.tx_hash)  
            await self.telegram.notify_order_filled(order.user_id, order, result)  
        else:  
            await self.orders.fail(order.id, result.error)  
            await self.telegram.notify_order_failed(order.user_id, order,  
result.error)
```

```
    except Exception as e:  
        await self.orders.fail(order.id, str(e))  
        self.monitor.log_error("order_execution", e)
```

```
    async def position_monitoring_loop(self):  
        while self.running:  
            try:  
                positions = await self.db.get_all_open_positions()
```



```

for position in positions:
    await self.check_position(position)

except Exception as e:
    self.monitor.log_error("position_monitoring", e)

await asyncio.sleep(1)

async def check_position(self, position):
    current_price = await self.market.get_price(position.token)

    if position.stop_loss and current_price <= position.stop_loss:
        await self.execute_stop_loss(position)

    if position.take_profit and current_price >= position.take_profit:
        await self.execute_take_profit(position)

```

SECTION 2: ORDER MANAGEMENT SYSTEM

Orders flow through multiple states. Proper state management prevents lost orders and double executions.

Order States:

- pending: Order created, awaiting processing
- validating: Risk checks in progress
- queued: Approved, waiting for execution
- executing: Transaction submitted
- confirming: Transaction mined, confirming
- completed: Successfully executed
- failed: Execution failed
- cancelled: User cancelled
- expired: Time limit exceeded

Order Manager:

```

from enum import Enum
from dataclasses import dataclass
from datetime import datetime

```

```
from typing import Optional
import asyncio
```

```
class OrderStatus(Enum):
    PENDING = "pending"
    VALIDATING = "validating"
    QUEUED = "queued"
    EXECUTING = "executing"
    CONFIRMING = "confirming"
    COMPLETED = "completed"
    FAILED = "failed"
    CANCELLED = "cancelled"
    EXPIRED = "expired"
```

```
class OrderType(Enum):
    MARKET_BUY = "market_buy"
    MARKET_SELL = "market_sell"
    LIMIT_BUY = "limit_buy"
    LIMIT_SELL = "limit_sell"
    SNIPE = "snipe"
    STOP_LOSS = "stop_loss"
    TAKE_PROFIT = "take_profit"
```

```
@dataclass
class Order:
    id: str
    user_id: int
    order_type: OrderType
    token_address: str
    amount: int
    price_limit: Optional[float]
    slippage: float
    gas_price: Optional[int]
    status: OrderStatus
    created_at: datetime
    updated_at: datetime
    tx_hash: Optional[str]
    error: Optional[str]
    metadata: dict
```

```
class OrderManager:
```

```
    def __init__(self, database):
        self.db = database
        self.pending_queue = asyncio.Queue()
        self.order_locks = {}
```

```
    async def create_order(self, user_id, order_type, token, amount,
**kwargs):
```

```
        order_id = self.generate_order_id()
```

```
        order = Order(
            id = order_id,
            user_id = user_id,
            order_type = OrderType(order_type),
            token_address = token,
            amount = amount,
            price_limit = kwargs.get("price_limit"),
            slippage = kwargs.get("slippage", 0.5),
            gas_price = kwargs.get("gas_price"),
            status = OrderStatus.PENDING,
            created_at = datetime.utcnow(),
            updated_at = datetime.utcnow(),
            tx_hash = None,
            error = None,
            metadata = kwargs.get("metadata", {})
        )
```

```
        await self.db.execute("""
INSERT INTO orders
(id, user_id, order_type, token_address, amount, price_limit,
slippage, gas_price, status, created_at, updated_at, metadata)
VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9, $10, $11, $12)
""", order.id, order.user_id, order.order_type.value,
order.token_address,
order.amount, order.price_limit, order.slippage, order.gas_price,
order.status.value, order.created_at, order.updated_at,
order.metadata)
```

```
        await self.pending_queue.put(order)
```

return order

```
async def get_pending(self, limit=100):
    rows = await self.db.fetch("""
SELECT * FROM orders
WHERE status IN ('pending', 'queued')
ORDER BY created_at ASC
LIMIT $1
""", limit)
```

return [self._row_to_order(row) for row in rows]

```
async def update_status(self, order_id, status, **kwargs):
    async with self._get_lock(order_id):
        updates = ["status = $2", "updated_at = NOW()"]
        params = [order_id, status.value]
        param_idx = 3
```

```
    if "tx_hash" in kwargs:
        updates.append(f"tx_hash = ${param_idx}")
        params.append(kwargs["tx_hash"])
        param_idx += 1
```

```
    if "error" in kwargs:
        updates.append(f"error = ${param_idx}")
        params.append(kwargs["error"])
        param_idx += 1
```

```
    query = f"UPDATE orders SET {' '.join(updates)} WHERE id = $1"
    await self.db.execute(query, *params)
```

```
async def complete(self, order_id, tx_hash):
    await self.update_status(
        order_id,
        OrderStatus.COMPLETED,
        tx_hash=tx_hash
    )
```

```
async def fail(self, order_id, error):
```

```
await self.update_status(  
    order_id,  
    OrderStatus.FAILED,  
    error = error  
)
```

```
async def reject(self, order_id, reason):  
    await self.update_status(  
        order_id,  
        OrderStatus.FAILED,  
        error = f'Rejected: {reason}'  
    )
```

```
async def cancel(self, order_id, user_id):  
    order = await self.get_order(order_id)
```

```
    if order.user_id != user_id:  
        raise ValueError("Not authorized to cancel this order")
```

```
    if order.status not in [OrderStatus.PENDING,  
        OrderStatus.QUEUED]:  
        raise ValueError("Cannot cancel order in current state")
```

```
    await self.update_status(order_id, OrderStatus.CANCELLED)
```

```
    async def get_order(self, order_id):  
        row = await self.db.fetchrow(  
            "SELECT * FROM orders WHERE id = $1",  
            order_id  
        )
```

```
    if not row:  
        return None
```

```
    return self._row_to_order(row)
```

```
    async def get_user_orders(self, user_id, status = None, limit = 50):  
        if status:  
            rows = await self.db.fetch("""  
            SELECT * FROM orders
```

```

WHERE user_id = $1 AND status = $2
ORDER BY created_at DESC
LIMIT $3
""" , user_id, status.value, limit)
else:
rows = await self.db.fetch("""
SELECT * FROM orders
WHERE user_id = $1
ORDER BY created_at DESC
LIMIT $2
""" , user_id, limit)

return [self._row_to_order(row) for row in rows]

```

```

def generate_order_id(self):
import uuid
return str(uuid.uuid4())[:8]

```

```

def _row_to_order(self, row):
return Order(
id = row["id"],
user_id = row["user_id"],
order_type = OrderType(row["order_type"]),
token_address = row["token_address"],
amount = row["amount"],
price_limit = row["price_limit"],
slippage = row["slippage"],
gas_price = row["gas_price"],
status = OrderStatus(row["status"]),
created_at = row["created_at"],
updated_at = row["updated_at"],
tx_hash = row["tx_hash"],
error = row["error"],
metadata = row["metadata"] or {}
)

```

```

def _get_lock(self, order_id):
if order_id not in self.order_locks:
self.order_locks[order_id] = asyncio.Lock()
return self.order_locks[order_id]

```

SECTION 3: EXECUTION ENGINE

The execution engine submits transactions to the blockchain. Speed and reliability are critical.

Execution Engine:

```
class ExecutionEngine:
```

```
    def __init__(self, rpc_config):
        self.rpc_config = rpc_config
        self.providers = []
        self.nonce_manager = NonceManager()
        self.gas_oracle = GasOracle()
        self.running = False
```

```
    async def start(self):
        self.running = True
```

```
    for rpc_url in self.rpc_config.urls:
        provider = RPCProvider(rpc_url)
        await provider.connect()
        self.providers.append(provider)
```

```
    await self.gas_oracle.start()
```

```
    async def stop(self):
        self.running = False
```

```
    for provider in self.providers:
        await provider.disconnect()
```

```
    await self.gas_oracle.stop()
```

```
    async def execute(self, order):
        try:
            user_wallet = await self.get_user_wallet(order.user_id)
```

```
tx_params = await self.build_transaction(order, user_wallet)
```

```
signed_tx = await self.sign_transaction(tx_params, user_wallet)
```

```
tx_hash = await self.submit_transaction(signed_tx)
```

```
receipt = await self.wait_for_confirmation(tx_hash)
```

```
if receipt.status == 1:
    return ExecutionResult(
        success=True,
        tx_hash=tx_hash,
        gas_used=receipt.gas_used,
        effective_price=self.calculate_effective_price(receipt, order)
    )
else:
    return ExecutionResult(
        success=False,
        tx_hash=tx_hash,
        error="Transaction reverted"
    )
```

```
except Exception as e:
    return ExecutionResult(
        success=False,
        error=str(e)
    )
```

```
async def build_transaction(self, order, wallet):
    gas_price = order.gas_price or await self.gas_oracle.get_gas_price()
```

```
    nonce = await self.nonce_manager.get_nonce(wallet.address)
```

```
    if order.order_type in [OrderType.MARKET_BUY,
        OrderType.SNIPE]:
        tx_data = await self.build_buy_transaction(order, wallet)
    elif order.order_type == OrderType.MARKET_SELL:
        tx_data = await self.build_sell_transaction(order, wallet)
    else:
        raise ValueError(f"Unsupported order type: {order.order_type}")
```



```

return {
    "to": tx_data["to"],
    "value": tx_data["value"],
    "data": tx_data["data"],
    "gas": tx_data["gas"],
    "gasPrice": gas_price,
    "nonce": nonce,
    "chainId": self.rpc_config.chain_id
}

```

```

async def build_buy_transaction(self, order, wallet):
    router = self.get_dex_router(order.token_address)

```

```

    path = await router.get_buy_path(order.token_address)

```

```

    min_output = await self.calculate_min_output(
        order.amount,
        path,
        order.slippage
    )

```

```

    deadline = int(time.time()) + 300

```

```

    return {
        "to": router.address,
        "value": order.amount,
        "data": router.encode_swap(
            amount_in=order.amount,
            amount_out_min=min_output,
            path=path,
            recipient=wallet.address,
            deadline=deadline
        ),
        "gas": 300000
    }

```

```

async def build_sell_transaction(self, order, wallet):
    router = self.get_dex_router(order.token_address)

```

```
path = await router.get_sell_path(order.token_address)
```

```
token_balance = await self.get_token_balance(  
order.token_address,  
wallet.address  
)
```

```
if order.amount > token_balance:  
raise ValueError("Insufficient token balance")
```

```
min_output = await self.calculate_min_output(  
order.amount,  
path,  
order.slippage  
)
```

```
deadline = int(time.time()) + 300
```

```
return {  
"to": router.address,  
"value": 0,  
"data": router.encode_swap(  
amount_in = order.amount,  
amount_out_min = min_output,  
path = path,  
recipient = wallet.address,  
deadline = deadline  
),  
"gas": 350000  
}
```

```
async def submit_transaction(self, signed_tx):  
for provider in self.providers:  
try:  
tx_hash = await provider.send_transaction(signed_tx)  
return tx_hash  
except Exception as e:  
continue
```

```
raise Exception("All RPC providers failed")
```

```

async def wait_for_confirmation(self, tx_hash, timeout=60):
    start_time = time.time()

    while time.time() - start_time < timeout:
        for provider in self.providers:
            try:
                receipt = await provider.get_transaction_receipt(tx_hash)
            if receipt:
                return receipt
            except Exception:
                continue

        await asyncio.sleep(1)

    raise Exception("Transaction confirmation timeout")

    def get_dex_router(self, token_address):
        return self.rpc_config.default_router

    async def get_user_wallet(self, user_id):
        pass

    async def sign_transaction(self, tx_params, wallet):
        pass

    async def calculate_min_output(self, amount_in, path, slippage):
        pass

    async def get_token_balance(self, token, wallet):
        pass

    def calculate_effective_price(self, receipt, order):
        pass

    @dataclass
    class ExecutionResult:
        success: bool
        tx_hash: str = None

```

```
gas_used: int = None
effective_price: float = None
error: str = None
```

SECTION 4: TOKEN SNIPIING

Sniping buys tokens immediately when they launch or list. Speed is everything - milliseconds matter.

Snipe Monitor:

```
class SnipeMonitor:
```

```
    def __init__(self, rpc_provider, database):
        self.rpc = rpc_provider
        self.db = database
        self.pending_snipes = {}
        self.running = False
```

```
    async def start(self):
        self.running = True
```

```
    asyncio.create_task(self.monitor_new_pairs())
    asyncio.create_task(self.monitor_liquidity_adds())
```

```
    async def stop(self):
        self.running = False
```

```
    async def add_snipe(self, user_id, token, config):
        snipe_id = f'{user_id}:{token}:{time.time()}'
```

```
        self.pending_snipes[snipe_id] = {
            "user_id": user_id,
            "token": token,
            "amount": config["amount"],
            "max_price": config.get("max_price"),
            "min_liquidity": config.get("min_liquidity", 1000),
            "slippage": config.get("slippage", 10),
            "gas_multiplier": config.get("gas_multiplier", 1.5),
```

```
"auto_sell": config.get("auto_sell"),  
"created_at": time.time(),  
"status": "pending"  
}
```

```
return snipe_id
```

```
async def cancel_snipe(self, snipe_id, user_id):  
    snipe = self.pending_snipes.get(snipe_id)
```

```
    if not snipe:  
        return False
```

```
    if snipe["user_id"] != user_id:  
        return False
```

```
    del self.pending_snipes[snipe_id]  
    return True
```

```
async def monitor_new_pairs(self):  
    factory_address = "0x..."
```

```
    async for event in self.rpc.subscribe_events(factory_address,  
"PairCreated"):
```

```
        token0 = event["args"]["token0"]  
        token1 = event["args"]["token1"]  
        pair = event["args"]["pair"]
```

```
        await self.handle_new_pair(token0, token1, pair)
```

```
    async def handle_new_pair(self, token0, token1, pair):  
        target_token = None
```

```
        base_tokens = ["0xweth", "0xusdt", "0xusdc"]
```

```
        if token0 in base_tokens:  
            target_token = token1  
        elif token1 in base_tokens:  
            target_token = token0  
        else:
```

return

```
matching_snipes = [  
(sid, snipe) for sid, snipe in self.pending_snipes.items()  
if snipe["token"].lower() == target_token.lower()  
and snipe["status"] == "pending"  
]
```

```
for snipe_id, snipe in matching_snipes:  
    asyncio.create_task(self.execute_snipe(snipe_id, snipe, pair))
```

```
async def monitor_liquidity_adds(self):  
    router_address = "0x..."
```

```
    async for event in self.rpc.subscribe_events(router_address,  
"AddLiquidity"):  
        token = event["args"]["token"]  
        liquidity_amount = event["args"]["amount"]
```

```
        await self.handle_liquidity_add(token, liquidity_amount)
```

```
    async def handle_liquidity_add(self, token, liquidity_amount):  
        matching_snipes = [  
        (sid, snipe) for sid, snipe in self.pending_snipes.items()  
        if snipe["token"].lower() == token.lower()  
        and snipe["status"] == "pending"  
        ]
```

```
        for snipe_id, snipe in matching_snipes:  
            if liquidity_amount >= snipe["min_liquidity"]:  
                asyncio.create_task(self.execute_snipe(snipe_id, snipe, None))
```

```
    async def execute_snipe(self, snipe_id, snipe, pair):  
        snipe["status"] = "executing"
```

```
        try:  
            is_safe = await self.safety_check(snipe["token"])
```

```
        if not is_safe:  
            snipe["status"] = "rejected"
```

```
await self.notify_snipe_rejected(snipe, "Failed safety check")
return
```

```
if snipe["max_price"]:
    current_price = await self.get_token_price(snipe["token"])
    if current_price > snipe["max_price"]:
        snipe["status"] = "rejected"
        await self.notify_snipe_rejected(snipe, "Price exceeded max")
    return
```

```
gas_price = await self.get_fast_gas_price()
gas_price = int(gas_price * snipe["gas_multiplier"])
```

```
tx_hash = await self.submit_buy(
    user_id=snipe["user_id"],
    token=snipe["token"],
    amount=snipe["amount"],
    slippage=snipe["slippage"],
    gas_price=gas_price
)
```

```
snipe["status"] = "executed"
snipe["tx_hash"] = tx_hash
```

```
if snipe["auto_sell"]:
    await self.setup_auto_sell(snipe)
```

```
await self.notify_snipe_executed(snipe)
```

```
except Exception as e:
    snipe["status"] = "failed"
    snipe["error"] = str(e)
    await self.notify_snipe_failed(snipe, str(e))
```

```
finally:
    if snipe_id in self.pending_snipes:
        del self.pending_snipes[snipe_id]
```

```
async def safety_check(self, token):
    is_honeypot = await self.check_honeypot(token)
```

```
if is_honeypot:  
    return False
```

```
tax = await self.get_token_tax(token)  
if tax > 15:  
    return False
```

```
is_renounced = await self.check_ownership_renounced(token)  
  
return True
```

```
async def check_honeypot(self, token):  
    return False
```

```
async def get_token_tax(self, token):  
    return 5
```

```
async def check_ownership_renounced(self, token):  
    return True
```

```
async def get_token_price(self, token):  
    return 0.001
```

```
async def get_fast_gas_price(self):  
    return 50000000000
```

```
async def submit_buy(self, user_id, token, amount, slippage,  
gas_price):  
    return "0x" + "a" * 64
```

```
async def setup_auto_sell(self, snipe):  
    pass
```

```
async def notify_snipe_executed(self, snipe):  
    pass
```

```
async def notify_snipe_rejected(self, snipe, reason):  
    pass
```

```
async def notify_snipe_failed(self, snipe, error):
```


pass

SECTION 5: COPY TRADING

Copy trading follows successful wallets automatically. When they buy, you buy. When they sell, you sell.

Copy Trading Engine:

class CopyTradingEngine:

```
def __init__(self, rpc_provider, execution_engine, database):
    self.rpc = rpc_provider
    self.executor = execution_engine
    self.db = database
    self.tracked_wallets = {}
    self.user_subscriptions = {}
    self.running = False

    async def start(self):
        self.running = True

        await self.load_subscriptions()

        asyncio.create_task(self.monitor_wallets())

    async def stop(self):
        self.running = False

    async def load_subscriptions(self):
        rows = await self.db.fetch("""
        SELECT * FROM copy_subscriptions WHERE is_active = TRUE
        """)

        for row in rows:
            wallet = row["tracked_wallet"]
            user_id = row["user_id"]

            if wallet not in self.tracked_wallets:
```

```
self.tracked_wallets[wallet] = {  
    "subscribers": [],  
    "last_tx": row.get("last_processed_tx")  
}
```

```
self.tracked_wallets[wallet]["subscribers"].append({  
    "user_id": user_id,  
    "copy_percentage": row["copy_percentage"],  
    "max_per_trade": row["max_per_trade"],  
    "copy_buys": row["copy_buys"],  
    "copy_sells": row["copy_sells"],  
    "min_trade_size": row["min_trade_size"]  
})
```

```
async def subscribe(self, user_id, wallet_address, config):  
    await self.db.execute("""  
    INSERT INTO copy_subscriptions  
    (user_id, tracked_wallet, copy_percentage, max_per_trade,  
    copy_buys, copy_sells, min_trade_size, is_active, created_at)  
    VALUES ($1, $2, $3, $4, $5, $6, $7, TRUE, NOW())  
    ON CONFLICT (user_id, tracked_wallet) DO UPDATE SET  
    copy_percentage = $3,  
    max_per_trade = $4,  
    is_active = TRUE  
    """, user_id, wallet_address, config["copy_percentage"],  
    config["max_per_trade"], config.get("copy_buys", True),  
    config.get("copy_sells", True), config.get("min_trade_size", 0))
```

```
if wallet_address not in self.tracked_wallets:  
    self.tracked_wallets[wallet_address] = {  
        "subscribers": [],  
        "last_tx": None  
    }
```

```
self.tracked_wallets[wallet_address]["subscribers"].append({  
    "user_id": user_id,  
    "copy_percentage": config["copy_percentage"],  
    "max_per_trade": config["max_per_trade"],  
    "copy_buys": config.get("copy_buys", True),  
    "copy_sells": config.get("copy_sells", True),
```

```
"min_trade_size": config.get("min_trade_size", 0)
})
```

```
async def unsubscribe(self, user_id, wallet_address):
    await self.db.execute("""
    UPDATE copy_subscriptions
    SET is_active = FALSE
    WHERE user_id = $1 AND tracked_wallet = $2
    """, user_id, wallet_address)
```

```
if wallet_address in self.tracked_wallets:
    self.tracked_wallets[wallet_address]["subscribers"] = [
        s for s in self.tracked_wallets[wallet_address]["subscribers"]
        if s["user_id"] != user_id
    ]
```

```
async def monitor_wallets(self):
    while self.running:
        for wallet, data in list(self.tracked_wallets.items()):
            try:
                await self.check_wallet_activity(wallet, data)
            except Exception as e:
                print(f"Error monitoring {wallet}: {e}")
```

```
await asyncio.sleep(2)
```

```
async def check_wallet_activity(self, wallet, data):
    transactions = await self.rpc.get_transactions(wallet, limit=10)
```

```
new_txs = []
for tx in transactions:
    if data["last_tx"] and tx["hash"] <= data["last_tx"]:
        break
    new_txs.append(tx)
```

```
new_txs.reverse()
```

```
for tx in new_txs:
    trade = await self.parse_trade(tx)
```

```
if trade:
    await self.process_trade(wallet, trade, data["subscribers"])
```

```
data["last_tx"] = tx["hash"]
```

```
async def parse_trade(self, tx):
    method = self.decode_method(tx["input"])
```

```
if not method:
    return None
```

```
swap_methods = [
    "swapExactETHForTokens",
    "swapExactTokensForETH",
    "swapExactTokensForTokens"
]
```

```
if method["name"] not in swap_methods:
    return None
```

```
return {
    "type": "buy" if "ForTokens" in method["name"] else "sell",
    "token": method["args"]["path"][-1] if "ForTokens" in
method["name"] else method["args"]["path"][0],
    "amount_in": method["args"]["amountIn"] if "amountIn" in
method["args"] else tx["value"],
    "tx_hash": tx["hash"]
}
```

```
async def process_trade(self, whale_wallet, trade, subscribers):
    for subscriber in subscribers:
        if trade["type"] == "buy" and not subscriber["copy_buys"]:
            continue
        if trade["type"] == "sell" and not subscriber["copy_sells"]:
            continue
```

```
if trade["amount_in"] < subscriber["min_trade_size"]:
    continue
```

```
copy_amount = int(trade["amount_in"] *
```

```
subscriber["copy_percentage"] / 100)
copy_amount = min(copy_amount, subscriber["max_per_trade"])
```

```
asyncio.create_task(
    self.execute_copy_trade(
        subscriber["user_id"],
        trade,
        copy_amount,
        whale_wallet
    )
)
```

```
async def execute_copy_trade(self, user_id, original_trade, amount,
                             whale_wallet):
```

```
    try:
        if original_trade["type"] == "buy":
            order_type = OrderType.MARKET_BUY
        else:
            order_type = OrderType.MARKET_SELL
```

```
        await self.executor.execute_order(
            user_id=user_id,
            order_type=order_type,
            token=original_trade["token"],
            amount=amount,
            metadata={
                "copy_from": whale_wallet,
                "original_tx": original_trade["tx_hash"]
            }
        )
```

```
        await self.notify_copy_executed(user_id, original_trade, amount)
```

```
    except Exception as e:
        await self.notify_copy_failed(user_id, original_trade, str(e))
```

```
def decode_method(self, input_data):
    return None
```

```
async def notify_copy_executed(self, user_id, trade, amount):
```

pass

```
async def notify_copy_failed(self, user_id, trade, error):  
    pass
```

SECTION 6: MEV PROTECTION

MEV (Maximal Extractable Value) attacks frontrun your transactions. Protection is essential.

MEV Protection Strategies:

```
class MEVProtection:
```

```
    def __init__(self, config):  
        self.config = config  
        self.flashbots_relay = config.flashbots_relay  
        self.private_pool = config.private_pool
```

```
    async def submit_protected_transaction(self, signed_tx,  
        protection_level="standard"):  
        if protection_level == "flashbots":  
            return await self.submit_via_flashbots(signed_tx)
```

```
        elif protection_level == "private":  
            return await self.submit_via_private_pool(signed_tx)
```

```
        elif protection_level == "timing":  
            return await self.submit_with_timing_protection(signed_tx)
```

```
    else:  
        return await self.submit_standard(signed_tx)
```

```
    async def submit_via_flashbots(self, signed_tx):  
        bundle = {  
            "signedTransactions": [signed_tx],  
            "blockNumber": await self.get_next_block(),  
            "minTimestamp": 0,  
            "maxTimestamp": int(time.time()) + 120
```

```
}
```

```
response = await self.flashbots_relay.send_bundle(bundle)
```

```
return response["bundleHash"]
```

```
async def submit_via_private_pool(self, signed_tx):
```

```
response = await self.private_pool.submit_transaction(signed_tx)
```

```
return response["txHash"]
```

```
async def submit_with_timing_protection(self, signed_tx):
```

```
block_time = 12
```

```
current_block = await self.get_current_block()
```

```
block_timestamp = current_block["timestamp"]
```

```
time_in_block = time.time() - block_timestamp
```

```
if time_in_block < block_time * 0.8:
```

```
delay = (block_time * 0.85) - time_in_block
```

```
await asyncio.sleep(delay)
```

```
return await self.submit_standard(signed_tx)
```

```
async def submit_standard(self, signed_tx):
```

```
pass
```

```
async def get_next_block(self):
```

```
current = await self.get_current_block()
```

```
return current["number"] + 1
```

```
async def get_current_block(self):
```

```
pass
```

```
def calculate_sandwich_risk(self, trade_amount, pool_liquidity):
```

```
impact = trade_amount / pool_liquidity
```

```
if impact > 0.01:
```

```
return "high"
```

```
elif impact > 0.005:
```

```
return "medium"  
else:  
return "low"
```

```
def recommend_protection(self, trade_amount, pool_liquidity):  
risk = self.calculate_sandwich_risk(trade_amount, pool_liquidity)
```

```
if risk == "high":  
return "flashbots"  
elif risk == "medium":  
return "private"  
else:  
return "standard"
```

SECTION 7: RISK MANAGEMENT

Risk management protects users from catastrophic losses. Never skip this.

Risk Manager:

```
class RiskManager:
```

```
def __init__(self, config, database):  
self.config = config  
self.db = database
```

```
self.max_position_size_percent = config.get("max_position_percent",  
25)  
self.max_daily_loss_percent = config.get("max_daily_loss_percent",  
10)  
self.max_single_trade_percent =  
config.get("max_single_trade_percent", 10)  
self.min_liquidity_usd = config.get("min_liquidity_usd", 10000)  
self.blocked_tokens = set(config.get("blocked_tokens", []))
```

```
async def validate_order(self, order):  
checks = [  
self.check_token_blocked(order),
```



```
self.check_position_size(order),
self.check_daily_loss_limit(order),
self.check_single_trade_limit(order),
self.check_liquidity(order),
self.check_token_safety(order)
]
```

```
for check in checks:
    result = await check
```

```
if not result.approved:
    return result
```

```
return ValidationResult(approved = True)
```

```
async def check_token_blocked(self, order):
    if order.token_address.lower() in self.blocked_tokens:
        return ValidationResult(
            approved = False,
            reason = "Token is blocked"
        )
```

```
return ValidationResult(approved = True)
```

```
async def check_position_size(self, order):
    user_balance = await self.get_user_balance(order.user_id)
```

```
if order.order_type in [OrderType.MARKET_BUY,
OrderType.SNIPE]:
    trade_value = order.amount
else:
    trade_value = await self.get_position_value(
        order.user_id,
        order.token_address,
        order.amount
    )
```

```
position_percent = (trade_value / user_balance) * 100
```

```
if position_percent > self.max_position_size_percent:
```

```
return ValidationResult(
    approved = False,
    reason = f"Position size {position_percent:.1f}% exceeds max
{self.max_position_size_percent}%"
)
```

```
return ValidationResult(approved = True)
```

```
async def check_daily_loss_limit(self, order):
    daily_pnl = await self.get_daily_pnl(order.user_id)
    user_balance = await self.get_user_balance(order.user_id)
```

```
    daily_loss_percent = (-daily_pnl / user_balance) * 100 if daily_pnl
    < 0 else 0
```

```
    if daily_loss_percent >= self.max_daily_loss_percent:
        return ValidationResult(
            approved = False,
            reason = f"Daily loss limit reached ({daily_loss_percent:.1f}%"
        )
```

```
    return ValidationResult(approved = True)
```

```
async def check_single_trade_limit(self, order):
    user_balance = await self.get_user_balance(order.user_id)
```

```
    trade_percent = (order.amount / user_balance) * 100
```

```
    if trade_percent > self.max_single_trade_percent:
        return ValidationResult(
            approved = False,
            reason = f"Trade size {trade_percent:.1f}% exceeds max
{self.max_single_trade_percent}%"
        )
```

```
    return ValidationResult(approved = True)
```

```
async def check_liquidity(self, order):
    liquidity_usd = await self.get_token_liquidity(order.token_address)
```

```
if liquidity_usd < self.min_liquidity_usd:  
    return ValidationResult(  
        approved = False,  
        reason = f"Insufficient liquidity: ${liquidity_usd:,.0f}"  
    )
```

```
trade_value_usd = await self.get_trade_value_usd(order)
```

```
if trade_value_usd > liquidity_usd * 0.05:  
    return ValidationResult(  
        approved = False,  
        reason = "Trade too large relative to liquidity"  
    )
```

```
return ValidationResult(approved = True)
```

```
async def check_token_safety(self, order):  
    if order.order_type == OrderType.MARKET_SELL:  
        return ValidationResult(approved = True)
```

```
is_honeypot = await self.check_honeypot(order.token_address)
```

```
if is_honeypot:  
    return ValidationResult(  
        approved = False,  
        reason = "Token appears to be a honeypot"  
    )
```

```
tax = await self.get_token_tax(order.token_address)
```

```
if tax > 20:  
    return ValidationResult(  
        approved = False,  
        reason = f"Token tax too high: {tax}%"  
    )
```

```
return ValidationResult(approved = True)
```

```
async def get_user_balance(self, user_id):  
    row = await self.db.fetchrow(  

```

```
"SELECT balance FROM user_balances WHERE user_id = $1",
user_id
)
return row["balance"] if row else 0
```

```
async def get_daily_pnl(self, user_id):
row = await self.db.fetchrow("""
SELECT COALESCE(SUM(pnl), 0) as daily_pnl
FROM trades
WHERE user_id = $1
AND created_at >= CURRENT_DATE
""", user_id)
return row["daily_pnl"]
```

```
async def get_position_value(self, user_id, token, amount):
return amount
```

```
async def get_token_liquidity(self, token):
return 100000
```

```
async def get_trade_value_usd(self, order):
return order.amount
```

```
async def check_honeypot(self, token):
return False
```

```
async def get_token_tax(self, token):
return 5
```

```
@dataclass
class ValidationResult:
    approved: bool
    reason: str = None
```

SECTION 8: POSITION MANAGEMENT

Track open positions, calculate PnL, and manage exits.

Position Manager:

class PositionManager:

```
def __init__(self, database, market_data):
```

```
    self.db = database
```

```
    self.market = market_data
```

```
    async def open_position(self, user_id, token, entry_amount,  
entry_price, tx_hash):
```

```
        position_id = self.generate_position_id()
```

```
        await self.db.execute("""
```

```
INSERT INTO positions
```

```
(id, user_id, token_address, entry_amount, current_amount,  
entry_price, entry_value, stop_loss, take_profit, status,  
opened_at, tx_hash)
```

```
VALUES ($1, $2, $3, $4, $4, $5, $6, NULL, NULL, 'open', NOW(),  
$7)
```

```
""", position_id, user_id, token, entry_amount, entry_price,  
entry_amount * entry_price, tx_hash)
```

```
return position_id
```

```
    async def close_position(self, position_id, exit_amount, exit_price,  
tx_hash):
```

```
        position = await self.get_position(position_id)
```

```
        if not position:
```

```
            raise ValueError("Position not found")
```

```
        exit_value = exit_amount * exit_price
```

```
        entry_value = position["entry_value"]
```

```
        pnl = exit_value - entry_value
```

```
        pnl_percent = (pnl / entry_value) * 100
```

```
        await self.db.execute("""
```

```
UPDATE positions SET
```

```
status = 'closed',
```

```
exit_amount = $1,  
exit_price = $2,  
exit_value = $3,  
pnl = $4,  
pnl_percent = $5,  
closed_at = NOW(),  
exit_tx_hash = $6  
WHERE id = $7  
""", exit_amount, exit_price, exit_value, pnl, pnl_percent, tx_hash,  
position_id)
```

```
await self.db.execute("""  
INSERT INTO trades  
(user_id, position_id, token, entry_price, exit_price,  
amount, pnl, pnl_percent, created_at)  
VALUES ($1, $2, $3, $4, $5, $6, $7, $8, NOW())  
""", position["user_id"], position_id, position["token_address"],  
position["entry_price"], exit_price, exit_amount, pnl, pnl_percent)
```

```
return {"pnl": pnl, "pnl_percent": pnl_percent}
```

```
async def set_stop_loss(self, position_id, stop_price):  
await self.db.execute(  
"UPDATE positions SET stop_loss = $1 WHERE id = $2",  
stop_price, position_id  
)
```

```
async def set_take_profit(self, position_id, take_profit_price):  
await self.db.execute(  
"UPDATE positions SET take_profit = $1 WHERE id = $2",  
take_profit_price, position_id  
)
```

```
async def get_position(self, position_id):  
return await self.db.fetchrow(  
"SELECT * FROM positions WHERE id = $1",  
position_id  
)
```

```
async def get_open_positions(self, user_id):
```

```
return await self.db.fetch("""
SELECT * FROM positions
WHERE user_id = $1 AND status = 'open'
ORDER BY opened_at DESC
""", user_id)
```

```
async def get_portfolio_summary(self, user_id):
    positions = await self.get_open_positions(user_id)
```

```
    total_value = 0
    total_pnl = 0
```

```
    position_details = []
```

```
    for position in positions:
        current_price = await
self.market.get_price(position["token_address"])
        current_value = position["current_amount"] * current_price
```

```
    unrealized_pnl = current_value - position["entry_value"]
    unrealized_pnl_percent = (unrealized_pnl / position["entry_value"])
    * 100
```

```
    total_value += current_value
    total_pnl += unrealized_pnl
```

```
    position_details.append({
        "position_id": position["id"],
        "token": position["token_address"],
        "entry_price": position["entry_price"],
        "current_price": current_price,
        "amount": position["current_amount"],
        "value": current_value,
        "pnl": unrealized_pnl,
        "pnl_percent": unrealized_pnl_percent
    })
```

```
    return {
        "total_value": total_value,
        "total_pnl": total_pnl,
```

```

"total_pnl_percent": (total_pnl / (total_value - total_pnl)) * 100 if
total_value != total_pnl else 0,
"positions": position_details
}

```

```

def generate_position_id(self):
import uuid
return str(uuid.uuid4())[:12]

```

SECTION 9: TELEGRAM BOT INTERFACE

The user-facing Telegram interface for the trading bot.

Trading Bot Commands:

```

from telegram import Update, InlineKeyboardButton,
InlineKeyboardMarkup
from telegram.ext import (
    Application, CommandHandler, CallbackQueryHandler,
    ConversationHandler, MessageHandler, filters, ContextTypes
)

```

class TradingBotInterface:

```

def __init__(self, token, trading_core):
self.token = token
self.core = trading_core
self.application = Application.builder().token(token).build()
self.setup_handlers()

def setup_handlers(self):
self.application.add_handler(CommandHandler("start",
self.cmd_start))
self.application.add_handler(CommandHandler("buy",
self.cmd_buy))
self.application.add_handler(CommandHandler("sell",
self.cmd_sell))
self.application.add_handler(CommandHandler("snipe",
self.cmd_snipe))

```



```
self.application.add_handler(CommandHandler("positions",
self.cmd_positions))
self.application.add_handler(CommandHandler("portfolio",
self.cmd_portfolio))
self.application.add_handler(CommandHandler("copy",
self.cmd_copy))
self.application.add_handler(CommandHandler("settings",
self.cmd_settings))
self.application.add_handler(CallbackQueryHandler(self.handle_callback))
```

```
async def cmd_start(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
    user = update.effective_user
```

```
await self.core.register_user(user.id, user.username)
```

```
keyboard = [
    [
        InlineKeyboardButton("Buy", callback_data="menu:buy"),
        InlineKeyboardButton("Sell", callback_data="menu:sell")
    ],
    [
        InlineKeyboardButton("Snipe", callback_data="menu:snipe"),
        InlineKeyboardButton("Copy Trade", callback_data="menu:copy")
    ],
    [
        InlineKeyboardButton("Portfolio", callback_data="menu:portfolio"),
        InlineKeyboardButton("Settings", callback_data="menu:settings")
    ]
]
```

```
await update.message.reply_text(
f'Welcome to TradingBot, {user.first_name}!\n\n'
f'Use the menu below or commands:\n'
f'/buy [token] [amount] - Buy tokens\n'
f'/sell [token] [amount/%) - Sell tokens\n'
f'/snipe [token] [amount] - Set up snipe\n'
f'/positions - View open positions\n'
f'/portfolio - Portfolio summary",
reply_markup=InlineKeyboardMarkup(keyboard))
```

)

```
async def cmd_buy(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
    args = context.args
```

```
    if len(args) < 2:
        await update.message.reply_text(
            "Usage: /buy [token_address] [amount_in_eth]\n"
            "Example: /buy 0x... 0.1"
        )
    return
```

```
    token = args[0]
    amount = float(args[1])
    amount_wei = int(amount * 1e18)
```

```
    user_id = update.effective_user.id
```

```
    token_info = await self.core.get_token_info(token)
    quote = await self.core.get_buy_quote(token, amount_wei)
```

```
    keyboard = [
        [
            InlineKeyboardButton(
                "Confirm Buy",
                callback_data=f"confirm_buy:{token}:{amount_wei}"
            ),
            InlineKeyboardButton("Cancel", callback_data="cancel")
        ]
    ]
```

```
    await update.message.reply_text(
        f"<b> Buy Order </b>\n\n"
        f"Token: {token_info['symbol']}\n"
        f"Amount: {amount} ETH\n"
        f"Expected: {quote['amount_out']:.4f} {token_info['symbol']}\n"
        f"Price Impact: {quote['price_impact']:.2f}%\n\n"
        f"Confirm this trade?",
        reply_markup=InlineKeyboardMarkup(keyboard),
```

```
parse_mode="HTML"  
)
```

```
async def cmd_sell(self, update: Update, context:  
ContextTypes.DEFAULT_TYPE):  
    args = context.args
```

```
    if len(args) < 2:  
        await update.message.reply_text(  
            "Usage: /sell [token_address] [amount or %]\n"  
            "Example: /sell 0x... 50%\n"  
            "Example: /sell 0x... 1000"  
        )  
    return
```

```
    token = args[0]  
    amount_str = args[1]
```

```
    user_id = update.effective_user.id  
    position = await self.core.get_position_by_token(user_id, token)
```

```
    if not position:  
        await update.message.reply_text("You don't have a position in this  
token.")  
    return
```

```
    if amount_str.endswith("%"):  
        percent = float(amount_str[:-1])  
        amount = int(position["current_amount"] * percent / 100)  
    else:  
        amount = int(float(amount_str))
```

```
    quote = await self.core.get_sell_quote(token, amount)
```

```
    keyboard = [  
        [  
            InlineKeyboardButton(  
                "Confirm Sell",  
                callback_data=f"confirm_sell:{token}:{amount}"  
            ),
```

```
InlineKeyboardButton("Cancel", callback_data="cancel")
]
```

```
await update.message.reply_text(
f'<b>Sell Order</b>\n\n'
f'Token: {position['symbol']}\n'
f'Amount: {amount}\n'
f'Expected: {quote['amount_out']:.4f} ETH\n'
f'Price Impact: {quote['price_impact']:.2f}%\n\n'
f'Confirm this trade?',
reply_markup=InlineKeyboardMarkup(keyboard),
parse_mode="HTML"
)
```

```
async def cmd_snipe(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
args = context.args
```

```
if len(args) < 2:
await update.message.reply_text(
"Usage: /snipe [token_address] [amount_eth]\n"
"Example: /snipe 0x... 0.5"
)
return
```

```
token = args[0]
amount = float(args[1])
```

```
user_id = update.effective_user.id
```

```
snipe_config = {
"amount": int(amount * 1e18),
"slippage": 10,
"gas_multiplier": 1.5
}
```

```
snipe_id = await self.core.snipe_monitor.add_snipe(
user_id, token, snipe_config
)
```

```

await update.message.reply_text(
f'<b> Snipe Created </b> \n\n'
f'Token: {token[:10]}...{token[-6:]} \n'
f'Amount: {amount} ETH \n'
f'Slippage: {snipe_config['slippage']}% \n'
f'Gas Multiplier: {snipe_config['gas_multiplier']}x \n\n'
f'Snipe ID: {snipe_id} \n'
f'Status: Waiting for liquidity...',
parse_mode="HTML"
)

```

```

async def cmd_positions(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
    user_id = update.effective_user.id

```

```

    positions = await
    self.core.position_manager.get_open_positions(user_id)

```

```

if not positions:
    await update.message.reply_text("You have no open positions.")
    return

```

```

text = "<b> Open Positions </b> \n\n"

```

```

for pos in positions:
    current_price = await
    self.core.market.get_price(pos["token_address"])
    pnl_percent = ((current_price - pos["entry_price"]) /
    pos["entry_price"]) * 100

```

```

emoji = "green_circle" if pnl_percent >= 0 else "red_circle"

```

```

text += (
f'{pos['symbol']} \n'
f'Entry: ${pos['entry_price']:.6f} \n'
f'Current: ${current_price:.6f} \n'
f'PnL: {pnl_percent: +.2f}% \n\n'
)

```

```
await update.message.reply_text(text, parse_mode="HTML")
```

```
async def cmd_portfolio(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
    user_id = update.effective_user.id
```

```
    summary = await
self.core.position_manager.get_portfolio_summary(user_id)
```

```
    text = (
f'<b>Portfolio Summary</b>\n\n'
f'Total Value: ${summary['total_value']:,.2f}\n'
f'Unrealized PnL: ${summary['total_pnl']: +,.2f}\n'
f'PnL %: {summary['total_pnl_percent']: +.2f}%\n\n'
f'Positions: {len(summary['positions'])}'
    )
```

```
await update.message.reply_text(text, parse_mode="HTML")
```

```
async def cmd_copy(self, update: Update, context:
ContextTypes.DEFAULT_TYPE):
    args = context.args
```

```
    if not args:
await update.message.reply_text(
"Usage: /copy [wallet_address] [percentage]\n"
"Example: /copy 0x... 50\n\n"
"/copy list - View active subscriptions\n"
"/copy stop [wallet] - Stop copying"
    )
```

```
    return
```

```
    if args[0] == "list":
await self.show_copy_subscriptions(update)
    elif args[0] == "stop" and len(args) > 1:
await self.stop_copy(update, args[1])
    else:
wallet = args[0]
percentage = int(args[1]) if len(args) > 1 else 50
```

```
await self.start_copy(update, wallet, percentage)
```

```
async def start_copy(self, update: Update, wallet, percentage):  
    user_id = update.effective_user.id
```

```
    config = {  
        "copy_percentage": percentage,  
        "max_per_trade": int(0.5 * 1e18),  
        "copy_buys": True,  
        "copy_sells": True  
    }
```

```
    await self.core.copy_engine.subscribe(user_id, wallet, config)
```

```
    await update.message.reply_text(  
        f'<b> Copy Trading Enabled </b> \n\n'  
        f'Wallet: {wallet[:10]}...{wallet[-6:]} \n'  
        f'Copy %: {percentage}% \n'  
        f'Max per trade: 0.5 ETH \n\n'  
        f'You will now copy this wallet's trades.',  
        parse_mode="HTML"  
    )
```

```
    async def handle_callback(self, update: Update, context:  
ContextTypes.DEFAULT_TYPE):  
        query = update.callback_query  
        await query.answer()
```

```
        data = query.data
```

```
        if data.startswith("confirm_buy:"):   
            parts = data.split(":")  
            token = parts[1]  
            amount = int(parts[2])  
            await self.execute_buy(query, token, amount)
```

```
        elif data.startswith("confirm_sell:"):   
            parts = data.split(":")  
            token = parts[1]  
            amount = int(parts[2])
```

```
await self.execute_sell(query, token, amount)
```

```
elif data == "cancel":
```

```
await query.message.edit_text("Cancelled.")
```

```
async def execute_buy(self, query, token, amount):
```

```
user_id = query.from_user.id
```

```
await query.message.edit_text("Executing buy order...")
```

```
try:
```

```
order = await self.core.create_order(
```

```
user_id=user_id,
```

```
order_type="market_buy",
```

```
token=token,
```

```
amount=amount
```

```
)
```

```
await query.message.edit_text(
```

```
f"<b>Order Submitted</b>\n\n"
```

```
f"Order ID: {order.id}\n"
```

```
f"Status: Processing...\n\n"
```

```
f"You will be notified when complete.",
```

```
parse_mode="HTML"
```

```
)
```

```
except Exception as e:
```

```
await query.message.edit_text(f"Order failed: {str(e)}")
```

```
async def execute_sell(self, query, token, amount):
```

```
user_id = query.from_user.id
```

```
await query.message.edit_text("Executing sell order...")
```

```
try:
```

```
order = await self.core.create_order(
```

```
user_id=user_id,
```

```
order_type="market_sell",
```

```
token=token,
```

```
amount=amount
```


)

```
await query.message.edit_text(
    f'<b>Order Submitted</b>\n\n'
    f'Order ID: {order.id}\n'
    f'Status: Processing...\n\n'
    f'You will be notified when complete.',
    parse_mode="HTML"
)
```

```
except Exception as e:
    await query.message.edit_text(f'Order failed: {str(e)}")
```

```
async def show_copy_subscriptions(self, update: Update):
    pass
```

```
async def stop_copy(self, update: Update, wallet):
    pass
```

```
def run(self):
    self.application.run_polling()
```

SECTION 10: PRODUCTION CONSIDERATIONS

Final considerations for production trading bots.

Production Checklist:

```
class ProductionReadiness:
```

```
def __init__(self):
    self.requirements = {
        "security": [
            "Private keys encrypted at rest",
            "API keys rotated regularly",
            "Rate limiting on all endpoints",
            "Input validation on all user data",
            "Audit logging enabled",
            "Penetration testing completed"
```

```

],
"reliability": [
    "Multiple RPC providers configured",
    "Automatic failover implemented",
    "Health checks on all services",
    "Graceful degradation for non-critical features",
    "Database backups automated",
    "Disaster recovery plan documented"
],
"monitoring": [
    "Transaction success rate tracked",
    "Latency metrics collected",
    "Error rates monitored",
    "Alerting configured for critical issues",
    "User balance reconciliation automated",
    "Audit trail for all trades"
],
"compliance": [
    "Terms of service published",
    "Risk disclaimers visible",
    "Trading limits documented",
    "Fee structure transparent",
    "User data handling documented"
]
}

```

```

def check_readiness(self, system):
    issues = []

```

```

    for category, items in self.requirements.items():
        for item in items:
            if not self.verify_requirement(system, item):
                issues.append(f'{category}: {item}')

```

```

    return {
        "ready": len(issues) == 0,
        "issues": issues
    }

```

```

def verify_requirement(self, system, item):

```

return True

CHAPTER SUMMARY

This chapter covered trading bot architecture:

Core components work together: Telegram interface, order management, execution engine, market data, risk management, and monitoring.

Order management tracks lifecycle from creation to completion. State machines prevent lost orders and double executions.

Execution engines optimize for speed and reliability. Multiple RPC providers, nonce management, and gas optimization.

Token sniping requires speed and safety checks. Monitor new pairs, verify liquidity, execute with high gas.

Copy trading follows successful wallets. Parse their trades, calculate proportional amounts, execute in parallel.

MEV protection prevents frontrunning. Flashbots bundles, private pools, and timing strategies.

Risk management prevents catastrophic losses. Position limits, daily loss limits, liquidity checks, token safety.

Position management tracks PnL and enables stop-loss/take-profit. Close tracking of entry, current value, and exits.

Production requires security, reliability, monitoring, and compliance. Checklists ensure nothing is missed.

Next chapter: Community Management. We automate Telegram groups at scale.

CHAPTER 7: COMMUNITY MANAGEMENT

Crypto communities live on Telegram. Thousands of members, 24/7

activity, endless spam attacks. Manual moderation is impossible. This chapter builds automated community management - welcome flows, anti-spam, raid protection, engagement tracking, and announcement systems. The same infrastructure that powers communities of 100,000+ members.

SECTION 1: COMMUNITY BOT ARCHITECTURE

Community bots handle different challenges than trading bots. High message volume. Spam attacks. Scammers. Group dynamics. The architecture must process thousands of messages per minute while maintaining responsiveness.

Core Components:

Message Processor: Receives all group messages, routes them through filters, triggers actions.

Member Manager: Tracks joins, leaves, bans. Manages verification flows.

Anti-Spam Engine: Detects and removes spam, scam links, raids.

Moderation System: Warn, mute, kick, ban. Track user infractions.

Engagement Tracker: Activity metrics, top contributors, sentiment analysis.

Announcement System: Scheduled posts, cross-group broadcasting.

Community Bot Core:

```
import asyncio
from datetime import datetime
from collections import defaultdict
from telegram import Update, ChatPermissions
from telegram.ext import Application, MessageHandler,
ChatMemberHandler, filters
```

```
class CommunityBot:
```

```
    def __init__(self, config, database):  
        self.config = config  
        self.db = database
```

```
        self.message_processor = MessageProcessor(self)  
        self.member_manager = MemberManager(self)  
        self.anti_spam = AntiSpamEngine(self)  
        self.moderation = ModerationSystem(self)  
        self.engagement = EngagementTracker(self)  
        self.announcements = AnnouncementSystem(self)
```

```
        self.application =  
        Application.builder().token(config.bot_token).build()  
        self.setup_handlers()
```

```
        self.group_settings = {}  
        self.running = False
```

```
    def setup_handlers(self):  
        self.application.add_handler(  
            MessageHandler(filters.ALL, self.handle_message)  
        )  
        self.application.add_handler(  
            ChatMemberHandler(self.handle_member_update)  
        )
```

```
    async def start(self):  
        self.running = True
```

```
        await self.load_group_settings()
```

```
        asyncio.create_task(self.anti_spam.start())  
        asyncio.create_task(self.announcements.start())
```

```
        await self.application.initialize()  
        await self.application.start()  
        await self.application.updater.start_polling()
```

```

async def stop(self):
    self.running = False

    await self.application.updater.stop()
    await self.application.stop()
    await self.application.shutdown()

    async def load_group_settings(self):
        rows = await self.db.fetch("SELECT * FROM group_settings")

        for row in rows:
            self.group_settings[row["group_id"]] = {
                "welcome_enabled": row["welcome_enabled"],
                "welcome_message": row["welcome_message"],
                "captcha_enabled": row["captcha_enabled"],
                "anti_spam_level": row["anti_spam_level"],
                "slow_mode": row["slow_mode"],
                "link_filter": row["link_filter"],
                "forward_filter": row["forward_filter"]
            }

    def get_settings(self, group_id):
        return self.group_settings.get(group_id, self.default_settings())

    def default_settings(self):
        return {
            "welcome_enabled": True,
            "welcome_message": "Welcome {user}!",
            "captcha_enabled": True,
            "anti_spam_level": "medium",
            "slow_mode": 0,
            "link_filter": True,
            "forward_filter": True
        }

    async def handle_message(self, update: Update, context):
        if not update.message:
            return

        message = update.message

```

```
chat_id = message.chat_id
user = message.from_user
```

```
if message.chat.type not in ["group", "supergroup"]:
    return
```

```
spam_result = await self.anti_spam.check_message(message)
```

```
if spam_result.is_spam:
    await self.handle_spam(message, spam_result)
    return
```

```
await self.message_processor.process(message)
await self.engagement.record_activity(chat_id, user.id, message)
```

```
async def handle_member_update(self, update: Update, context):
    member_update = update.chat_member
```

```
    if member_update.new_chat_member.status == "member":
        await self.member_manager.handle_join(
            member_update.chat.id,
            member_update.new_chat_member.user
        )
```

```
    elif member_update.new_chat_member.status == "left":
        await self.member_manager.handle_leave(
            member_update.chat.id,
            member_update.new_chat_member.user
        )
```

```
async def handle_spam(self, message, spam_result):
    try:
        await message.delete()
    except Exception:
        pass
```

```
await self.moderation.record_infraction(
    message.chat_id,
    message.from_user.id,
    "spam",
```

```
spam_result.reason
)
```

```
infractions = await self.moderation.get_infraction_count(
    message.chat_id,
    message.from_user.id
)
```

```
if infractions >= 3:
    await self.moderation.mute_user(
        message.chat_id,
        message.from_user.id,
        duration_minutes=60
    )
```

SECTION 2: MEMBER MANAGEMENT

New members need verification. Bots flood groups constantly. CAPTCHA challenges filter humans from bots.

Member Manager:

```
class MemberManager:
```

```
    def __init__(self, bot):
        self.bot = bot
        self.pending_verifications = {}
        self.verification_timeout = 300
```

```
    async def handle_join(self, chat_id, user):
        settings = self.bot.get_settings(chat_id)
```

```
        await self.record_join(chat_id, user)
```

```
        if settings["captcha_enabled"]:
            await self.start_verification(chat_id, user)
        else:
            if settings["welcome_enabled"]:
                await self.send_welcome(chat_id, user,
```



```

settings["welcome_message"])

async def start_verification(self, chat_id, user):
    await self.restrict_user(chat_id, user.id)

    captcha = self.generate_captcha()

    self.pending_verifications[f'{chat_id}:{user.id}'] = {
        "answer": captcha["answer"],
        "attempts": 0,
        "expires": datetime.utcnow().timestamp() +
self.verification_timeout
    }

    keyboard = []
    row = []

    for option in captcha["options"]:
        row.append(InlineKeyboardButton(
            option,
            callback_data = f'verify:{chat_id}:{user.id}:{option}'
        ))
        if len(row) == 3:
            keyboard.append(row)
            row = []

    if row:
        keyboard.append(row)

    message = await self.bot.application.bot.send_message(
        chat_id=chat_id,
        text=f'Welcome {user.mention_html()}!\n\n'
        f'Please solve this to verify you're human:\n\n'
        f'{captcha['question']}\n\n'
        f'You have 5 minutes.',
        reply_markup=InlineKeyboardMarkup(keyboard),
        parse_mode="HTML"
    )

    asyncio.create_task(

```

```
self.verification_timeout_handler(chat_id, user.id,  
message.message_id)  
)
```

```
async def verify_answer(self, chat_id, user_id, answer):  
    key = f"{chat_id}:{user_id}"  
    verification = self.pending_verifications.get(key)
```

```
if not verification:  
    return False
```

```
if datetime.utcnow().timestamp() > verification["expires"]:  
    await self.kick_user(chat_id, user_id)  
    del self.pending_verifications[key]  
    return False
```

```
if str(answer) == str(verification["answer"]):  
    del self.pending_verifications[key]  
    await self.unrestrict_user(chat_id, user_id)
```

```
settings = self.bot.get_settings(chat_id)  
if settings["welcome_enabled"]:  
    user = await self.bot.application.bot.get_chat_member(chat_id,  
user_id)  
    await self.send_welcome(chat_id, user.user,  
settings["welcome_message"])
```

```
return True  
else:  
    verification["attempts"] += 1
```

```
if verification["attempts"] >= 3:  
    await self.kick_user(chat_id, user_id)  
    del self.pending_verifications[key]
```

```
return False
```

```
async def verification_timeout_handler(self, chat_id, user_id,  
message_id):  
    await asyncio.sleep(self.verification_timeout)
```

```
key = f"{chat_id}:{user_id}"
```

```
if key in self.pending_verifications:  
del self.pending_verifications[key]  
await self.kick_user(chat_id, user_id)
```

```
try:  
await self.bot.application.bot.delete_message(chat_id, message_id)  
except Exception:  
pass
```

```
def generate_captcha(self):  
import random
```

```
captcha_types = ["math", "emoji", "word"]  
captcha_type = random.choice(captcha_types)
```

```
if captcha_type == "math":  
a = random.randint(1, 10)  
b = random.randint(1, 10)  
answer = a + b  
wrong_answers = [answer + random.randint(1, 5), answer -  
random.randint(1, 5)]
```

```
return {  
"question": f"What is {a} + {b}?",  
"answer": str(answer),  
"options": [str(x) for x in sorted([answer] + wrong_answers)]  
}
```

```
elif captcha_type == "emoji":  
emojis = ["dog", "cat", "bird", "fish"]  
target = random.choice(emojis)
```

```
return {  
"question": f"Select the {target}:",  
"answer": target,  
"options": emojis  
}
```

```
else:
    words = ["apple", "banana", "orange", "grape"]
    target = random.choice(words)
```

```
return {
    "question": f'Select '{target}':',
    "answer": target,
    "options": words
}
```

```
async def restrict_user(self, chat_id, user_id):
    permissions = ChatPermissions(
        can_send_messages=False,
        can_send_media_messages=False,
        can_send_polls=False,
        can_send_other_messages=False,
        can_add_web_page_previews=False
    )
```

```
await self.bot.application.bot.restrict_chat_member(
    chat_id, user_id, permissions
)
```

```
async def unrestrict_user(self, chat_id, user_id):
    permissions = ChatPermissions(
        can_send_messages=True,
        can_send_media_messages=True,
        can_send_polls=True,
        can_send_other_messages=True,
        can_add_web_page_previews=True,
        can_change_info=False,
        can_invite_users=True,
        can_pin_messages=False
    )
```

```
await self.bot.application.bot.restrict_chat_member(
    chat_id, user_id, permissions
)
```

```
async def kick_user(self, chat_id, user_id):
    await self.bot.application.bot.ban_chat_member(chat_id, user_id)
    await asyncio.sleep(1)
    await self.bot.application.bot.unban_chat_member(chat_id, user_id)
```

```
async def send_welcome(self, chat_id, user, template):
    message = template.format(
        user = user.mention_html(),
        username = user.username or user.first_name,
        first_name = user.first_name,
        chat_title = await self.get_chat_title(chat_id)
    )
```

```
    await self.bot.application.bot.send_message(
        chat_id = chat_id,
        text = message,
        parse_mode = "HTML"
    )
```

```
async def get_chat_title(self, chat_id):
    chat = await self.bot.application.bot.get_chat(chat_id)
    return chat.title
```

```
async def record_join(self, chat_id, user):
    await self.bot.db.execute("""
INSERT INTO member_events (group_id, user_id, event_type,
created_at)
VALUES ($1, $2, 'join', NOW())
""", chat_id, user.id)
```

```
async def handle_leave(self, chat_id, user):
    await self.bot.db.execute("""
INSERT INTO member_events (group_id, user_id, event_type,
created_at)
VALUES ($1, $2, 'leave', NOW())
""", chat_id, user.id)
```

SECTION 3: ANTI-SPAM ENGINE

Spam comes in waves. Link spam, forward spam, repetitive messages, raid attacks. Multi-layered defense stops them.

Anti-Spam Engine:

```
class AntiSpamEngine:
```

```
    def __init__(self, bot):
        self.bot = bot
```

```
    self.message_history = defaultdict(list)
    self.user_scores = defaultdict(float)
```

```
    self.spam_patterns = [
        r"t\\.me/(?!joinchat)",
        r"bit\\.ly",
        r"earn.*crypto",
        r"free.*giveaway",
        r"send.*eth.*receive",
        r"airdrop.*claim",
        r"@[a-zA-Z0-9_] + bot\b"
    ]
```

```
    self.running = False
```

```
    async def start(self):
        self.running = True
        asyncio.create_task(self.cleanup_loop())
```

```
    async def cleanup_loop(self):
        while self.running:
            await asyncio.sleep(300)
```

```
    cutoff = datetime.utcnow().timestamp() - 600
```

```
    for key in list(self.message_history.keys()):
        self.message_history[key] = [
            m for m in self.message_history[key]
            if m["timestamp"] > cutoff
        ]
```

```
for key in list(self.user_scores.keys()):
    self.user_scores[key] *= 0.9
    if self.user_scores[key] < 0.1:
        del self.user_scores[key]
```

```
async def check_message(self, message):
    user_id = message.from_user.id
    chat_id = message.chat_id
    key = f'{chat_id}:{user_id}'
```

```
settings = self.bot.get_settings(chat_id)
level = settings["anti_spam_level"]
```

```
if level == "off":
    return SpamResult(is_spam=False)
```

```
score = 0
reasons = []
```

```
if await self.check_new_user(message):
    score += 0.3
    reasons.append("new_user")
```

```
if settings["link_filter"] and await self.check_links(message):
    score += 0.5
    reasons.append("suspicious_link")
```

```
if settings["forward_filter"] and message.forward_date:
    score += 0.3
    reasons.append("forwarded")
```

```
pattern_result = await self.check_patterns(message)
if pattern_result:
    score += 0.6
    reasons.append(f'pattern:{pattern_result}')
```

```
if await self.check_repetition(message, key):
    score += 0.4
    reasons.append("repetition")
```

```
if await self.check_flood(message, key):
    score += 0.5
    reasons.append("flood")
```

```
if await self.check_raid(chat_id):
    score += 0.2
    reasons.append("raid_mode")
```

```
self.user_scores[key] += score
```

```
total_score = self.user_scores[key]
```

```
thresholds = {
    "low": 2.0,
    "medium": 1.0,
    "high": 0.5,
    "paranoid": 0.3
}
```

```
threshold = thresholds.get(level, 1.0)
```

```
if total_score >= threshold:
    return SpamResult(
        is_spam=True,
        score=total_score,
        reason=" , ".join(reasons)
    )
```

```
self.record_message(message, key)
```

```
return SpamResult(is_spam=False, score=total_score)
```

```
async def check_new_user(self, message):
    user_id = message.from_user.id
    chat_id = message.chat_id
```

```
    member = await self.bot.application.bot.get_chat_member(chat_id,
user_id)
```



```
if not hasattr(member, 'joined_date') or not member.joined_date:  
    return True
```

```
join_age = (datetime.utcnow() -  
member.joined_date).total_seconds()  
return join_age < 300
```

```
async def check_links(self, message):  
    if not message.text and not message.caption:  
        return False
```

```
text = message.text or message.caption
```

```
import re
```

```
suspicious_patterns = [  
    r"https?:\/\/[^\s] +",  
    r"t\.me/[^\s] +",  
    r"@[a-zA-Z0-9_] + bot",  
]
```

```
for pattern in suspicious_patterns:  
    if re.search(pattern, text, re.IGNORECASE):  
        return True
```

```
return False
```

```
async def check_patterns(self, message):  
    if not message.text and not message.caption:  
        return None
```

```
text = message.text or message.caption
```

```
import re
```

```
for pattern in self.spam_patterns:  
    if re.search(pattern, text, re.IGNORECASE):  
        return pattern
```

```
return None
```

```
async def check_repetition(self, message, key):
    if not message.text:
        return False
```

```
text = message.text.lower().strip()
```

```
recent_messages = self.message_history.get(key, [])
```

```
duplicates = sum(1 for m in recent_messages if m["text"] == text)
```

```
return duplicates >= 2
```

```
async def check_flood(self, message, key):
    recent_messages = self.message_history.get(key, [])
```

```
cutoff = datetime.utcnow().timestamp() - 10
```

```
recent_count = sum(1 for m in recent_messages if m["timestamp"]
> cutoff)
```

```
return recent_count >= 5
```

```
async def check_raid(self, chat_id):
    recent_joins = await self.bot.db.fetchval("""
SELECT COUNT(*) FROM member_events
WHERE group_id = $1
AND event_type = 'join'
AND created_at > NOW() - INTERVAL '1 minute'
""", chat_id)
```

```
return recent_joins > 10
```

```
def record_message(self, message, key):
    self.message_history[key].append({
        "text": (message.text or "").lower().strip(),
        "timestamp": datetime.utcnow().timestamp()
    })
```

@dataclass

```
class SpamResult:
    is_spam: bool
    score: float = 0
    reason: str = None
```

SECTION 4: MODERATION SYSTEM

Moderators need tools. Warn, mute, kick, ban. Track history.
Escalate automatically.

Moderation System:

```
class ModerationSystem:

    def __init__(self, bot):
        self.bot = bot

    async def warn_user(self, chat_id, user_id, reason, warned_by):
        await self.record_infraction(chat_id, user_id, "warn", reason,
        warned_by)

        infractions = await self.get_infraction_count(chat_id, user_id)

        user = await self.bot.application.bot.get_chat_member(chat_id,
        user_id)

        await self.bot.application.bot.send_message(
            chat_id=chat_id,
            text=f"{user.user.mention_html()} has been warned.\n"
            f"Reason: {reason}\n"
            f"Total warnings: {infractions}",
            parse_mode="HTML"
        )

        if infractions >= 3:
            await self.mute_user(chat_id, user_id, duration_minutes=60)
        if infractions >= 5:
            await self.ban_user(chat_id, user_id, reason="Too many warnings")
```

```

    async def mute_user(self, chat_id, user_id, duration_minutes = None,
reason = None):
        permissions = ChatPermissions(can_send_messages = False)

        until_date = None
        if duration_minutes:
            until_date = datetime.utcnow().timestamp() + (duration_minutes
* 60)

        await self.bot.application.bot.restrict_chat_member(
chat_id, user_id, permissions, until_date = until_date
        )

        await self.record_infraction(chat_id, user_id, "mute", reason or
"Muted")

        user = await self.bot.application.bot.get_chat_member(chat_id,
user_id)

        duration_text = f"for {duration_minutes} minutes" if
duration_minutes else ""

        await self.bot.application.bot.send_message(
chat_id = chat_id,
text = f"{user.user.mention_html()} has been
muted{duration_text}.",
parse_mode = "HTML"
        )

    async def unmute_user(self, chat_id, user_id):
        permissions = ChatPermissions(
can_send_messages = True,
can_send_media_messages = True,
can_send_polls = True,
can_send_other_messages = True,
can_add_web_page_previews = True
        )

        await self.bot.application.bot.restrict_chat_member(
chat_id, user_id, permissions

```

)

```
user = await self.bot.application.bot.get_chat_member(chat_id,
user_id)
```

```
await self.bot.application.bot.send_message(
chat_id=chat_id,
text=f'{user.user.mention_html()} has been unmuted.',
parse_mode="HTML"
)
```

```
async def kick_user(self, chat_id, user_id, reason = None):
user = await self.bot.application.bot.get_chat_member(chat_id,
user_id)
```

```
await self.bot.application.bot.ban_chat_member(chat_id, user_id)
await asyncio.sleep(1)
await self.bot.application.bot.unban_chat_member(chat_id, user_id)
```

```
await self.record_infraction(chat_id, user_id, "kick", reason or
"Kicked")
```

```
await self.bot.application.bot.send_message(
chat_id=chat_id,
text=f'{user.user.mention_html()} has been kicked.\n'
f'Reason: {reason or 'No reason provided'}',
parse_mode="HTML"
)
```

```
async def ban_user(self, chat_id, user_id, reason = None,
duration_hours = None):
user = await self.bot.application.bot.get_chat_member(chat_id,
user_id)
```

```
until_date = None
if duration_hours:
until_date = datetime.utcnow().timestamp() + (duration_hours *
3600)
```

```
await self.bot.application.bot.ban_chat_member(
```

```
chat_id, user_id, until_date=until_date
)
```

```
await self.record_infraction(chat_id, user_id, "ban", reason or
"Banned")
```

```
duration_text = f"for {duration_hours} hours" if duration_hours
else " permanently"
```

```
await self.bot.application.bot.send_message(
chat_id=chat_id,
text=f'{user.user.mention_html()} has been
banned{duration_text}.\n'
f'Reason: {reason or 'No reason provided'}',
parse_mode="HTML"
)
```

```
async def unban_user(self, chat_id, user_id):
await self.bot.application.bot.unban_chat_member(chat_id, user_id)
```

```
async def record_infraction(self, chat_id, user_id, action_type,
reason, performed_by=None):
await self.bot.db.execute("""
INSERT INTO infractions
(group_id, user_id, action_type, reason, performed_by, created_at)
VALUES ($1, $2, $3, $4, $5, NOW())
""", chat_id, user_id, action_type, reason, performed_by)
```

```
async def get_infraction_count(self, chat_id, user_id, days=30):
count = await self.bot.db.fetchval("""
SELECT COUNT(*) FROM infractions
WHERE group_id = $1 AND user_id = $2
AND created_at > NOW() - INTERVAL '%s days'
""", % days, chat_id, user_id)
```

```
return count
```

```
async def get_user_history(self, chat_id, user_id):
rows = await self.bot.db.fetch("""
SELECT * FROM infractions
```

```
WHERE group_id = $1 AND user_id = $2
ORDER BY created_at DESC
LIMIT 20
""", chat_id, user_id)
```

return rows

```
async def clear_warnings(self, chat_id, user_id):
    await self.bot.db.execute("""
DELETE FROM infractions
WHERE group_id = $1 AND user_id = $2
AND action_type = 'warn'
""", chat_id, user_id)
```

SECTION 5: ENGAGEMENT TRACKING

Healthy communities have active members. Track who contributes, identify trends, reward engagement.

Engagement Tracker:

```
class EngagementTracker:
```

```
    def __init__(self, bot):
        self.bot = bot
        self.daily_stats = defaultdict(lambda: defaultdict(int))
```

```
    async def record_activity(self, chat_id, user_id, message):
        key = f'{chat_id}:{datetime.utcnow().date()}'
```

```
        self.daily_stats[key][user_id] += 1
```

```
        activity_type = self.classify_activity(message)
```

```
        await self.bot.db.execute("""
INSERT INTO activity_log
(group_id, user_id, activity_type, created_at)
VALUES ($1, $2, $3, NOW())
""", chat_id, user_id, activity_type)
```

```
await self.update_user_stats(chat_id, user_id)
```

```
def classify_activity(self, message):  
    if message.text:  
        if len(message.text) > 100:  
            return "long_message"  
        return "message"  
    if message.photo:  
        return "photo"  
    if message.video:  
        return "video"  
    if message.sticker:  
        return "sticker"  
    if message.voice:  
        return "voice"  
    return "other"
```

```
async def update_user_stats(self, chat_id, user_id):  
    await self.bot.db.execute("""  
    INSERT INTO user_stats (group_id, user_id, message_count,  
last_active)  
VALUES ($1, $2, 1, NOW())  
ON CONFLICT (group_id, user_id) DO UPDATE SET  
message_count = user_stats.message_count + 1,  
last_active = NOW()  
""", chat_id, user_id)
```

```
async def get_top_contributors(self, chat_id, days=7, limit=10):  
    rows = await self.bot.db.fetch("""  
    SELECT user_id, COUNT(*) as activity_count  
    FROM activity_log  
    WHERE group_id = $1  
    AND created_at > NOW() - INTERVAL '%s days'  
    GROUP BY user_id  
    ORDER BY activity_count DESC  
    LIMIT $2  
""", % days, chat_id, limit)
```

```
contributors = []
```



```

for row in rows:
    try:
        member = await self.bot.application.bot.get_chat_member(
            chat_id, row["user_id"]
        )
        contributors.append({
            "user_id": row["user_id"],
            "username": member.user.username,
            "first_name": member.user.first_name,
            "count": row["activity_count"]
        })
    except Exception:
        pass

```

```

return contributors

```

```

async def get_group_stats(self, chat_id, days=7):
    stats = await self.bot.db.fetchrow("""
    SELECT
    COUNT(DISTINCT user_id) as active_users,
    COUNT(*) as total_messages,
    COUNT(*) FILTER (WHERE activity_type = 'message') as
text_messages,
    COUNT(*) FILTER (WHERE activity_type = 'photo') as photos,
    COUNT(*) FILTER (WHERE activity_type = 'sticker') as stickers
    FROM activity_log
    WHERE group_id = $1
    AND created_at > NOW() - INTERVAL '%s days'
    """ % days, chat_id)

```

```

joins = await self.bot.db.fetchval("""
    SELECT COUNT(*) FROM member_events
    WHERE group_id = $1
    AND event_type = 'join'
    AND created_at > NOW() - INTERVAL '%s days'
    """ % days, chat_id)

```

```

leaves = await self.bot.db.fetchval("""
    SELECT COUNT(*) FROM member_events

```

```

WHERE group_id = $1
AND event_type = 'leave'
AND created_at > NOW() - INTERVAL '%s days'
      """ % days, chat_id)

```

```

return {
    "active_users": stats["active_users"],
    "total_messages": stats["total_messages"],
    "text_messages": stats["text_messages"],
    "photos": stats["photos"],
    "stickers": stats["stickers"],
    "joins": joins,
    "leaves": leaves,
    "net_growth": joins - leaves,
    "avg_messages_per_user": stats["total_messages"] /
max(stats["active_users"], 1)
}

```

```

async def get_hourly_activity(self, chat_id, days=7):
    rows = await self.bot.db.fetch("""
SELECT
EXTRACT(HOUR FROM created_at) as hour,
COUNT(*) as count
FROM activity_log
WHERE group_id = $1
AND created_at > NOW() - INTERVAL '%s days'
GROUP BY EXTRACT(HOUR FROM created_at)
ORDER BY hour
      """ % days, chat_id)

```

```

hourly = {i: 0 for i in range(24)}

```

```

for row in rows:
    hourly[int(row["hour"])] = row["count"]

```

```

return hourly

```

SECTION 6: ANNOUNCEMENT SYSTEM

Project updates need to reach everyone. Scheduled posts, cross-group broadcasting, pin management.

Announcement System:

```
class AnnouncementSystem:
```

```
    def __init__(self, bot):
        self.bot = bot
        self.scheduled_announcements = []
        self.running = False
```

```
    async def start(self):
        self.running = True
        await self.load_scheduled()
        asyncio.create_task(self.scheduler_loop())
```

```
    async def load_scheduled(self):
        rows = await self.bot.db.fetch("""
        SELECT * FROM scheduled_announcements
        WHERE status = 'pending'
        AND scheduled_for > NOW()
        ORDER BY scheduled_for ASC
        """)
```

```
        self.scheduled_announcements = [dict(row) for row in rows]
```

```
    async def scheduler_loop(self):
        while self.running:
            now = datetime.utcnow()
```

```
            due_announcements = [
                a for a in self.scheduled_announcements
                if datetime.fromisoformat(str(a["scheduled_for"])) <= now
            ]
```

```
            for announcement in due_announcements:
                await self.send_announcement(announcement)
                self.scheduled_announcements.remove(announcement)
```

```
await asyncio.sleep(10)
```

```
async def schedule_announcement(self, group_ids, message,  
scheduled_for, options=None):  
    options = options or {}
```

```
    announcement_id = await self.bot.db.fetchval("""  
    INSERT INTO scheduled_announcements  
    (group_ids, message, scheduled_for, pin_message, status, created_at)  
    VALUES ($1, $2, $3, $4, 'pending', NOW())  
    RETURNING id  
    """, group_ids, message, scheduled_for, options.get("pin", False))
```

```
    announcement = {  
        "id": announcement_id,  
        "group_ids": group_ids,  
        "message": message,  
        "scheduled_for": scheduled_for,  
        "pin_message": options.get("pin", False),  
        "status": "pending"  
    }
```

```
    self.scheduled_announcements.append(announcement)  
    self.scheduled_announcements.sort(key=lambda x:  
x["scheduled_for"])
```

```
    return announcement_id
```

```
async def send_announcement(self, announcement):  
    group_ids = announcement["group_ids"]  
    message = announcement["message"]  
    pin = announcement.get("pin_message", False)
```

```
    results = {"success": [], "failed": []}
```

```
    for group_id in group_ids:  
        try:  
            sent_message = await self.bot.application.bot.send_message(  
                chat_id=group_id,  
                text=message,
```

```
parse_mode = "HTML"  
)
```

```
if pin:  
    await self.bot.application.bot.pin_chat_message(  
        chat_id = group_id,  
        message_id = sent_message.message_id  
    )
```

```
results["success"].append(group_id)
```

```
except Exception as e:  
    results["failed"].append({  
        "group_id": group_id,  
        "error": str(e)  
    })
```

```
await self.bot.db.execute("""  
UPDATE scheduled_announcements  
SET status = 'sent', sent_at = NOW()  
WHERE id = $1  
""", announcement["id"])
```

```
return results
```

```
async def broadcast_now(self, group_ids, message, options = None):  
    announcement = {  
        "id": None,  
        "group_ids": group_ids,  
        "message": message,  
        "pin_message": options.get("pin", False) if options else False  
    }
```

```
return await self.send_announcement(announcement)
```

```
async def cancel_announcement(self, announcement_id):  
    await self.bot.db.execute("""  
UPDATE scheduled_announcements  
SET status = 'cancelled'  
WHERE id = $1
```

```
""", announcement_id)
```

```
self.scheduled_announcements = [  
    a for a in self.scheduled_announcements  
    if a["id"] != announcement_id  
]
```

```
async def get_scheduled(self, group_id=None):  
    if group_id:  
        return [  
            a for a in self.scheduled_announcements  
            if group_id in a["group_ids"]  
        ]  
    return self.scheduled_announcements
```

SECTION 7: ADMIN COMMANDS

Administrators control the bot through commands. Settings, moderation, statistics.

Admin Command Handler:

```
class AdminCommands:
```

```
    def __init__(self, bot):  
        self.bot = bot  
        self.setup_commands()
```

```
    def setup_commands(self):  
        app = self.bot.application
```

```
        app.add_handler(CommandHandler("settings", self.cmd_settings))  
        app.add_handler(CommandHandler("setwelcome",  
self.cmd_set_welcome))  
        app.add_handler(CommandHandler("antispam",  
self.cmd_antispam))  
        app.add_handler(CommandHandler("warn", self.cmd_warn))  
        app.add_handler(CommandHandler("mute", self.cmd_mute))  
        app.add_handler(CommandHandler("unmute", self.cmd_unmute))
```

```
app.add_handler(CommandHandler("kick", self.cmd_kick))
app.add_handler(CommandHandler("ban", self.cmd_ban))
app.add_handler(CommandHandler("unban", self.cmd_unban))
app.add_handler(CommandHandler("stats", self.cmd_stats))
app.add_handler(CommandHandler("top", self.cmd_top))
app.add_handler(CommandHandler("announce",
self.cmd_announce))
app.add_handler(CommandHandler("schedule", self.cmd_schedule))
```

```
async def is_admin(self, chat_id, user_id):
    member = await self.bot.application.bot.get_chat_member(chat_id,
user_id)
    return member.status in ["creator", "administrator"]
```

```
async def cmd_settings(self, update: Update, context):
    chat_id = update.effective_chat.id
    user_id = update.effective_user.id
```

```
if not await self.is_admin(chat_id, user_id):
    await update.message.reply_text("Admin only command.")
    return
```

```
settings = self.bot.get_settings(chat_id)
```

```
text = (
    "<b>Group Settings</b>\n\n"
    f"Welcome Message: {'ON' if settings['welcome_enabled'] else
'OFF'}\n"
    f"CAPTCHA: {'ON' if settings['captcha_enabled'] else 'OFF'}\n"
    f"Anti-Spam Level: {settings['anti_spam_level']}\n"
    f"Link Filter: {'ON' if settings['link_filter'] else 'OFF'}\n"
    f"Forward Filter: {'ON' if settings['forward_filter'] else 'OFF'}\n"
)
```

```
keyboard = [
    [
        InlineKeyboardButton(
            "Toggle Welcome",
            callback_data="settings:toggle_welcome"
        ),
```

```

InlineKeyboardButton(
    "Toggle CAPTCHA",
    callback_data = "settings:toggle_captcha"
),
[
    InlineKeyboardButton(
        "Anti-Spam Settings",
        callback_data = "settings:antispam"
    )
]
]

```

```

await update.message.reply_text(
    text,
    reply_markup = InlineKeyboardMarkup(keyboard),
    parse_mode = "HTML"
)

```

```

async def cmd_set_welcome(self, update: Update, context):
    chat_id = update.effective_chat.id
    user_id = update.effective_user.id

```

```

    if not await self.is_admin(chat_id, user_id):
        await update.message.reply_text("Admin only command.")
    return

```

```

    if not context.args:
        await update.message.reply_text(
            "Usage: /setwelcome [message]\n\n"
            "Variables:\n"
            "{user} - Mention user\n"
            "{username} - Username\n"
            "{first_name} - First name\n"
            "{chat_title} - Group name"
        )
    return

```

```

welcome_message = " ".join(context.args)

```



```
await self.bot.db.execute("""
UPDATE group_settings
SET welcome_message = $1
WHERE group_id = $2
""", welcome_message, chat_id)
```

```
self.bot.group_settings[chat_id]["welcome_message"] =
welcome_message
```

```
await update.message.reply_text("Welcome message updated.")
```

```
async def cmd_antispam(self, update: Update, context):
chat_id = update.effective_chat.id
user_id = update.effective_user.id
```

```
if not await self.is_admin(chat_id, user_id):
await update.message.reply_text("Admin only command.")
return
```

```
if not context.args:
await update.message.reply_text(
"Usage: /antispam [level]\n\n"
"Levels: off, low, medium, high, paranoid"
)
return
```

```
level = context.args[0].lower()
```

```
if level not in ["off", "low", "medium", "high", "paranoid"]:
await update.message.reply_text("Invalid level.")
return
```

```
await self.bot.db.execute("""
UPDATE group_settings
SET anti_spam_level = $1
WHERE group_id = $2
""", level, chat_id)
```

```
self.bot.group_settings[chat_id]["anti_spam_level"] = level
```

```
await update.message.reply_text(f'Anti-spam level set to: {level}')
```

```
async def cmd_warn(self, update: Update, context):  
    chat_id = update.effective_chat.id  
    user_id = update.effective_user.id
```

```
if not await self.is_admin(chat_id, user_id):  
    await update.message.reply_text("Admin only command.")  
    return
```

```
if not update.message.reply_to_message:  
    await update.message.reply_text("Reply to a message to warn the  
user.")  
    return
```

```
target_user = update.message.reply_to_message.from_user  
reason = " ".join(context.args) if context.args else "No reason  
provided"
```

```
await self.bot.moderation.warn_user(  
    chat_id, target_user.id, reason, user_id  
)
```

```
async def cmd_mute(self, update: Update, context):  
    chat_id = update.effective_chat.id  
    user_id = update.effective_user.id
```

```
if not await self.is_admin(chat_id, user_id):  
    await update.message.reply_text("Admin only command.")  
    return
```

```
if not update.message.reply_to_message:  
    await update.message.reply_text("Reply to a message to mute the  
user.")  
    return
```

```
target_user = update.message.reply_to_message.from_user
```

```
duration = 60  
if context.args:
```

```
try:
    duration = int(context.args[0])
except ValueError:
    pass
```

```
await self.bot.moderation.mute_user(
    chat_id, target_user.id, duration_minutes = duration
)
```

```
async def cmd_unmute(self, update: Update, context):
    chat_id = update.effective_chat.id
    user_id = update.effective_user.id
```

```
if not await self.is_admin(chat_id, user_id):
    await update.message.reply_text("Admin only command.")
    return
```

```
if not update.message.reply_to_message:
    await update.message.reply_text("Reply to a message to unmute the
user.")
    return
```

```
target_user = update.message.reply_to_message.from_user
```

```
await self.bot.moderation.unmute_user(chat_id, target_user.id)
```

```
async def cmd_kick(self, update: Update, context):
    chat_id = update.effective_chat.id
    user_id = update.effective_user.id
```

```
if not await self.is_admin(chat_id, user_id):
    await update.message.reply_text("Admin only command.")
    return
```

```
if not update.message.reply_to_message:
    await update.message.reply_text("Reply to a message to kick the
user.")
    return
```

```
target_user = update.message.reply_to_message.from_user
```

```
reason = " ".join(context.args) if context.args else None
```

```
await self.bot.moderation.kick_user(chat_id, target_user.id, reason)
```

```
async def cmd_ban(self, update: Update, context):
```

```
chat_id = update.effective_chat.id
```

```
user_id = update.effective_user.id
```

```
if not await self.is_admin(chat_id, user_id):
```

```
await update.message.reply_text("Admin only command.")
```

```
return
```

```
if not update.message.reply_to_message:
```

```
await update.message.reply_text("Reply to a message to ban the  
user.")
```

```
return
```

```
target_user = update.message.reply_to_message.from_user
```

```
reason = " ".join(context.args) if context.args else None
```

```
await self.bot.moderation.ban_user(chat_id, target_user.id, reason)
```

```
async def cmd_unban(self, update: Update, context):
```

```
chat_id = update.effective_chat.id
```

```
user_id = update.effective_user.id
```

```
if not await self.is_admin(chat_id, user_id):
```

```
await update.message.reply_text("Admin only command.")
```

```
return
```

```
if not context.args:
```

```
await update.message.reply_text("Usage: /unban [user_id]")
```

```
return
```

```
target_user_id = int(context.args[0])
```

```
await self.bot.moderation.unban_user(chat_id, target_user_id)
```

```
await update.message.reply_text("User unbanned.")
```

```
async def cmd_stats(self, update: Update, context):
```

```
chat_id = update.effective_chat.id
user_id = update.effective_user.id
```

```
if not await self.is_admin(chat_id, user_id):
    await update.message.reply_text("Admin only command.")
    return
```

```
days = 7
if context.args:
    try:
        days = int(context.args[0])
    except ValueError:
        pass
```

```
stats = await self.bot.engagement.get_group_stats(chat_id, days)
```

```
text = (
    f'<b>Group Statistics ({days} days)</b>\n\n'
    f'Active Users: {stats['active_users']}\n'
    f'Total Messages: {stats['total_messages'];}\n'
    f'Text Messages: {stats['text_messages'];}\n'
    f'Photos: {stats['photos'];}\n'
    f'Stickers: {stats['stickers'];}\n\n'
    f'New Members: {stats['joins']}\n'
    f'Left Members: {stats['leaves']}\n'
    f'Net Growth: {stats['net_growth']: + }\n\n'
    f'Avg Messages/User: {stats['avg_messages_per_user']:.1f}'
)
```

```
await update.message.reply_text(text, parse_mode="HTML")
```

```
async def cmd_top(self, update: Update, context):
    chat_id = update.effective_chat.id
```

```
days = 7
if context.args:
    try:
        days = int(context.args[0])
    except ValueError:
        pass
```

```
contributors = await self.bot.engagement.get_top_contributors(
chat_id, days=days, limit=10
)
```

```
if not contributors:
```

```
    await update.message.reply_text("No activity data available.")
    return
```

```
text = f"<b>Top Contributors ({days} days)</b>\n\n"
```

```
for i, contributor in enumerate(contributors, 1):
    name = contributor["username"] or contributor["first_name"]
    text += f"{i}. {name}: {contributor['count']} messages\n"
```

```
await update.message.reply_text(text, parse_mode="HTML")
```

```
async def cmd_announce(self, update: Update, context):
    chat_id = update.effective_chat.id
    user_id = update.effective_user.id
```

```
    if not await self.is_admin(chat_id, user_id):
        await update.message.reply_text("Admin only command.")
    return
```

```
    if not context.args:
        await update.message.reply_text(
            "Usage: /announce [message]\n"
            "Add --pin to pin the message"
        )
    return
```

```
    message_parts = " ".join(context.args)
    pin = "--pin" in message_parts
    message = message_parts.replace("--pin", "").strip()
```

```
    result = await self.bot.announcements.broadcast_now(
        [chat_id],
        message,
        {"pin": pin}
    )
```

```

)

if result["success"]:
    await update.message.reply_text("Announcement sent.")
else:
    await update.message.reply_text("Failed to send announcement.")

async def cmd_schedule(self, update: Update, context):
    chat_id = update.effective_chat.id
    user_id = update.effective_user.id

    if not await self.is_admin(chat_id, user_id):
        await update.message.reply_text("Admin only command.")
        return

    await update.message.reply_text(
        "Usage: /schedule [time] [message]\n\n"
        "Time format: YYYY-MM-DD HH:MM or +Xh (X hours from\n"
        "now)\n\n"
        "Example: /schedule +2h This is a scheduled message\n"
        "Example: /schedule 2024-01-15 14:00 Important update!"
    )

```

SECTION 8: RAID PROTECTION

Raids overwhelm groups with spam or bots. Automatic detection and lockdown.

Raid Protection:

```

class RaidProtection:

    def __init__(self, bot):
        self.bot = bot
        self.raid_mode = {}
        self.join_history = defaultdict(list)

    async def check_raid(self, chat_id):
        now = datetime.utcnow().timestamp()

```

```
cutoff = now - 60
```

```
self.join_history[chat_id] = [  
t for t in self.join_history[chat_id] if t > cutoff  
]
```

```
recent_joins = len(self.join_history[chat_id])
```

```
if recent_joins > 20:  
    await self.activate_raid_mode(chat_id, "mass_join")  
    return True
```

```
return False
```

```
async def record_join(self, chat_id):  
    self.join_history[chat_id].append(datetime.utcnow().timestamp())  
    await self.check_raid(chat_id)
```

```
async def activate_raid_mode(self, chat_id, reason):  
    if chat_id in self.raid_mode:  
        return
```

```
self.raid_mode[chat_id] = {  
    "activated_at": datetime.utcnow(),  
    "reason": reason  
}
```

```
await self.bot.application.bot.set_chat_permissions(  
    chat_id,  
    ChatPermissions(  
        can_send_messages=False,  
        can_send_media_messages=False,  
        can_send_polls=False,  
        can_send_other_messages=False,  
        can_add_web_page_previews=False,  
        can_change_info=False,  
        can_invite_users=False,  
        can_pin_messages=False  
    )  
)
```



```
await self.bot.application.bot.send_message(
chat_id = chat_id,
text = "<b>RAID DETECTED</b> \n\n"
"Group has been locked down.\n"
f"Reason: {reason}\n\n"
"Admins: Use /unlock to restore permissions.",
parse_mode = "HTML"
)
```

```
asyncio.create_task(self.auto_unlock(chat_id))
```

```
async def auto_unlock(self, chat_id):
await asyncio.sleep(300)
```

```
if chat_id in self.raid_mode:
await self.deactivate_raid_mode(chat_id)
```

```
async def deactivate_raid_mode(self, chat_id):
if chat_id not in self.raid_mode:
return
```

```
del self.raid_mode[chat_id]
```

```
await self.bot.application.bot.set_chat_permissions(
chat_id,
ChatPermissions(
can_send_messages = True,
can_send_media_messages = True,
can_send_polls = True,
can_send_other_messages = True,
can_add_web_page_previews = True,
can_change_info = False,
can_invite_users = True,
can_pin_messages = False
)
)
```

```
await self.bot.application.bot.send_message(
chat_id = chat_id,
```

```
text="Raid mode deactivated. Normal permissions restored.",  
parse_mode="HTML"  
)
```

SECTION 9: DATABASE SCHEMA

Database structure for community management.

PostgreSQL Schema:

```
schema_sql = ""
```

```
CREATE TABLE group_settings (  
    group_id BIGINT PRIMARY KEY,  
    welcome_enabled BOOLEAN DEFAULT TRUE,  
    welcome_message TEXT DEFAULT 'Welcome {user}!',  
    captcha_enabled BOOLEAN DEFAULT TRUE,  
    anti_spam_level VARCHAR(20) DEFAULT 'medium',  
    slow_mode INTEGER DEFAULT 0,  
    link_filter BOOLEAN DEFAULT TRUE,  
    forward_filter BOOLEAN DEFAULT TRUE,  
    created_at TIMESTAMP DEFAULT NOW(),  
    updated_at TIMESTAMP DEFAULT NOW()  
);
```

```
CREATE TABLE member_events (  
    id SERIAL PRIMARY KEY,  
    group_id BIGINT NOT NULL,  
    user_id BIGINT NOT NULL,  
    event_type VARCHAR(20) NOT NULL,  
    created_at TIMESTAMP DEFAULT NOW()  
);
```

```
CREATE INDEX idx_member_events_group ON  
member_events(group_id);  
CREATE INDEX idx_member_events_created ON  
member_events(created_at);
```

```
CREATE TABLE infractions (  
    id SERIAL PRIMARY KEY,  
    group_id BIGINT NOT NULL,  
    user_id BIGINT NOT NULL,  
    event_type VARCHAR(20) NOT NULL,  
    created_at TIMESTAMP DEFAULT NOW()  
);
```

```
id SERIAL PRIMARY KEY,  
group_id BIGINT NOT NULL,  
user_id BIGINT NOT NULL,  
action_type VARCHAR(20) NOT NULL,  
reason TEXT,  
performed_by BIGINT,  
created_at TIMESTAMP DEFAULT NOW()  
);
```

```
CREATE INDEX idx_infractions_user ON infractions(group_id,  
user_id);
```

```
CREATE TABLE activity_log (  
id SERIAL PRIMARY KEY,  
group_id BIGINT NOT NULL,  
user_id BIGINT NOT NULL,  
activity_type VARCHAR(20) NOT NULL,  
created_at TIMESTAMP DEFAULT NOW()  
);
```

```
CREATE INDEX idx_activity_group ON activity_log(group_id);  
CREATE INDEX idx_activity_created ON activity_log(created_at);
```

```
CREATE TABLE user_stats (  
group_id BIGINT NOT NULL,  
user_id BIGINT NOT NULL,  
message_count INTEGER DEFAULT 0,  
last_active TIMESTAMP,  
PRIMARY KEY (group_id, user_id)  
);
```

```
CREATE TABLE scheduled_announcements (  
id SERIAL PRIMARY KEY,  
group_ids BIGINT[] NOT NULL,  
message TEXT NOT NULL,  
scheduled_for TIMESTAMP NOT NULL,  
pin_message BOOLEAN DEFAULT FALSE,  
status VARCHAR(20) DEFAULT 'pending',  
created_at TIMESTAMP DEFAULT NOW(),  
sent_at TIMESTAMP
```

```
);
```

```
CREATE INDEX idx_announcements_scheduled ON  
scheduled_announcements(scheduled_for)  
WHERE status = 'pending';
```

```
"""
```

SECTION 10: COMPLETE BOT ASSEMBLY

Putting all components together.

Complete Community Bot:

```
class CompleteCommunityBot:
```

```
    def __init__(self, config):
```

```
        self.config = config
```

```
        self.db = None
```

```
        self.application =
```

```
Application.builder().token(config.bot_token).build()
```

```
        self.member_manager = MemberManager(self)
```

```
        self.anti_spam = AntiSpamEngine(self)
```

```
        self.moderation = ModerationSystem(self)
```

```
        self.engagement = EngagementTracker(self)
```

```
        self.announcements = AnnouncementSystem(self)
```

```
        self.raid_protection = RaidProtection(self)
```

```
        self.admin_commands = AdminCommands(self)
```

```
        self.group_settings = {}
```

```
    async def connect_database(self):
```

```
        import asyncpg
```

```
        self.db = await asyncpg.create_pool(
```

```
            self.config.database_url,
```

```
min_size = 5,  
max_size = 20  
)
```

```
async def start(self):  
    await self.connect_database()  
    await self.load_group_settings()
```

```
    await self.anti_spam.start()  
    await self.announcements.start()
```

```
self.setup_handlers()
```

```
    await self.application.initialize()  
    await self.application.start()  
    await self.application.updater.start_polling(  
        allowed_updates = ["message", "callback_query", "chat_member"]  
    )
```

```
def setup_handlers(self):  
    self.application.add_handler(  
        MessageHandler(  
            filters.ALL & ~filters.COMMAND,  
            self.handle_message  
        ),  
        group = 0  
    )
```

```
    self.application.add_handler(  
        ChatMemberHandler(  
            self.handle_member_update,  
            ChatMemberHandler.CHAT_MEMBER  
        ),  
        group = 0  
    )
```

```
    self.application.add_handler(  
        CallbackQueryHandler(self.handle_callback)  
    )
```

```
async def handle_message(self, update: Update, context):
```

```
if not update.message:
    return

message = update.message
chat_id = message.chat_id

if message.chat.type not in ["group", "supergroup"]:
    return

spam_result = await self.anti_spam.check_message(message)

if spam_result.is_spam:
    try:
        await message.delete()
    except Exception:
        pass

    await self.moderation.record_infraction(
        chat_id,
        message.from_user.id,
        "spam",
        spam_result.reason
    )
    return

await self.engagement.record_activity(
    chat_id,
    message.from_user.id,
    message
)

async def handle_member_update(self, update: Update, context):
    member_update = update.chat_member
    chat_id = member_update.chat.id

    if member_update.new_chat_member.status == "member":
        await self.raid_protection.record_join(chat_id)
        await self.member_manager.handle_join(
            chat_id,
            member_update.new_chat_member.user
```

)

```
elif member_update.new_chat_member.status == "left":
    await self.member_manager.handle_leave(
        chat_id,
        member_update.new_chat_member.user
    )
```

```
async def handle_callback(self, update: Update, context):
    query = update.callback_query
    await query.answer()
```

```
data = query.data
```

```
if data.startswith("verify:"):
    parts = data.split(":")
    chat_id = int(parts[1])
    user_id = int(parts[2])
    answer = parts[3]
```

```
if query.from_user.id != user_id:
    return
```

```
result = await self.member_manager.verify_answer(chat_id,
    user_id, answer)
```

```
if result:
    await query.message.delete()
else:
    await query.message.edit_text("Wrong answer. Try again.")
```

```
elif data.startswith("settings:"):
    action = data.split(":")[1]
    await self.handle_settings_callback(query, action)
```

```
async def handle_settings_callback(self, query, action):
    chat_id = query.message.chat_id
    user_id = query.from_user.id
```

```
member = await self.application.bot.get_chat_member(chat_id,
```

```

user_id)
if member.status not in ["creator", "administrator"]:
    return

settings = self.get_settings(chat_id)

if action == "toggle_welcome":
    new_value = not settings["welcome_enabled"]
    await self.db.execute(
        "UPDATE group_settings SET welcome_enabled = $1 WHERE
group_id = $2",
        new_value, chat_id
    )
    self.group_settings[chat_id]["welcome_enabled"] = new_value

elif action == "toggle_captcha":
    new_value = not settings["captcha_enabled"]
    await self.db.execute(
        "UPDATE group_settings SET captcha_enabled = $1 WHERE
group_id = $2",
        new_value, chat_id
    )
    self.group_settings[chat_id]["captcha_enabled"] = new_value

settings = self.get_settings(chat_id)

text = (
    "<b>Group Settings</b>\n\n"
    f"Welcome Message: {'ON' if settings['welcome_enabled'] else
'OFF'}\n"
    f"CAPTCHA: {'ON' if settings['captcha_enabled'] else 'OFF'}\n"
    f"Anti-Spam Level: {settings['anti_spam_level']}\n"
)

await query.message.edit_text(text, parse_mode="HTML")

async def load_group_settings(self):
    rows = await self.db.fetch("SELECT * FROM group_settings")

    for row in rows:

```



```
self.group_settings[row["group_id"]] = dict(row)
```

```
def get_settings(self, group_id):
    if group_id not in self.group_settings:
        return {
            "welcome_enabled": True,
            "welcome_message": "Welcome {user}!",
            "captcha_enabled": True,
            "anti_spam_level": "medium",
            "link_filter": True,
            "forward_filter": True
        }
    return self.group_settings[group_id]
```

```
async def main():
    from dataclasses import dataclass
```

```
@dataclass
class Config:
    bot_token: str
    database_url: str
```

```
config = Config(
    bot_token="YOUR_BOT_TOKEN",
    database_url="postgresql://user:pass@localhost/community"
)
```

```
bot = CompleteCommunityBot(config)
await bot.start()
```

```
try:
    while True:
        await asyncio.sleep(1)
except KeyboardInterrupt:
    pass
```

```
if __name__ == "__main__":
    asyncio.run(main())
```

CHAPTER SUMMARY

This chapter covered community management automation:

Community bots handle high-volume message processing, spam detection, and member management across multiple groups.

Member management includes CAPTCHA verification for new users, welcome messages, and tracking joins/leaves.

Anti-spam engines use multi-layered detection: pattern matching, flood detection, link filtering, and user reputation scoring.

Moderation systems provide warn, mute, kick, ban with automatic escalation based on infraction history.

Engagement tracking identifies top contributors, measures activity patterns, and provides group health metrics.

Announcement systems support scheduled posts, cross-group broadcasting, and pinning.

Admin commands give moderators control over settings, moderation actions, and statistics.

Raid protection automatically locks down groups during coordinated spam attacks.

Database schemas track members, infractions, activity, and scheduled content.

Next chapter: Security and Best Practices. We protect bots, users, and funds.

CHAPTER 8: SECURITY AND BEST PRACTICES

Telegram bots handling cryptocurrency face real threats. Private keys stolen. User funds drained. Bots hijacked. This chapter covers security from every angle - key management, input validation, rate

limiting, secure communication, audit logging, and production hardening. One vulnerability can destroy everything you've built.

SECTION 1: PRIVATE KEY SECURITY

Private keys control funds. Their security is paramount. Never store keys in plaintext. Never log them. Never expose them.

Key Management:

```
import os
import base64
from cryptography.fernet import Fernet
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import
PBKDF2HMAC

class KeyManager:

    def __init__(self, master_password, salt=None):
        self.salt = salt or os.urandom(16)
        self.cipher = self._create_cipher(master_password)

    def _create_cipher(self, password):
        kdf = PBKDF2HMAC(
            algorithm=hashes.SHA256(),
            length=32,
            salt=self.salt,
            iterations=480000,
        )

        key = base64.urlsafe_b64encode(kdf.derive(password.encode()))
        return Fernet(key)

    def encrypt_key(self, private_key):
        encrypted = self.cipher.encrypt(private_key.encode())
        return base64.b64encode(encrypted).decode()

    def decrypt_key(self, encrypted_key):
```

```
encrypted = base64.b64decode(encrypted_key.encode())
decrypted = self.cipher.decrypt(encrypted)
return decrypted.decode()
```

```
def generate_wallet(self):
    from eth_account import Account
```

```
    account = Account.create()
```

```
    encrypted_key = self.encrypt_key(account.key.hex())
```

```
    return {
        "address": account.address,
        "encrypted_key": encrypted_key
    }
```

```
def get_account(self, encrypted_key):
    from eth_account import Account
```

```
    private_key = self.decrypt_key(encrypted_key)
    return Account.from_key(private_key)
```

Hardware Security Module Integration:

```
class HSMKeyManager:
```

```
    def __init__(self, hsm_config):
        self.hsm = self._connect_hsm(hsm_config)
```

```
    def _connect_hsm(self, config):
        pass
```

```
    async def sign_transaction(self, tx_data, key_id):
        signature = await self.hsm.sign(
            key_id=key_id,
            data=tx_data,
            algorithm="secp256k1"
        )
```

return signature

```
async def create_key(self, user_id):
    key_id = await self.hsm.generate_key(
        algorithm="secp256k1",
        label=f"user_{user_id}"
    )
```

```
public_key = await self.hsm.get_public_key(key_id)
```

```
address = self._derive_address(public_key)
```

```
return {
    "key_id": key_id,
    "address": address
}
```

```
def _derive_address(self, public_key):
    from eth_utils import keccak, to_checksum_address
```

```
    address_bytes = keccak(public_key)[-20:]
    return to_checksum_address(address_bytes)
```

Environment Variable Security:

```
import os
from dataclasses import dataclass
```

```
@dataclass
class SecureConfig:
    bot_token: str
    database_url: str
    encryption_key: str
    rpc_url: str
```

```
@classmethod
def from_environment(cls):
    required_vars = [
        "BOT_TOKEN",
```

```
"DATABASE_URL",  
"ENCRYPTION_KEY",  
"RPC_URL"  
]
```

```
for var in required_vars:  
    if var not in os.environ:  
        raise ValueError(f'Missing required environment variable: {var}')
```

```
return cls(  
    bot_token=os.environ["BOT_TOKEN"],  
    database_url=os.environ["DATABASE_URL"],  
    encryption_key=os.environ["ENCRYPTION_KEY"],  
    rpc_url=os.environ["RPC_URL"]  
)
```

```
def _repr_(self):  
    return "SecureConfig(***)"
```

```
def _str_(self):  
    return "SecureConfig(***)"
```

SECTION 2: INPUT VALIDATION

All user input is hostile. Validate everything. Sanitize before use.

Input Validator:

```
import re  
from typing import Optional
```

```
class InputValidator:
```

```
    ETH_ADDRESS_PATTERN = re.compile(r"^0x[a-fA-F0-9]{40}$")  
    TX_HASH_PATTERN = re.compile(r"^0x[a-fA-F0-9]{64}$")  
    AMOUNT_PATTERN = re.compile(r"^[0-9]+(\.[0-9]+)?$")
```

```
    @classmethod  
    def validate_eth_address(cls, address: str) -> Optional[str]:
```

```
if not address:  
    return None
```

```
address = address.strip()
```

```
if not cls.ETH_ADDRESS_PATTERN.match(address):  
    return None
```

```
try:  
    from eth_utils import to_checksum_address, is_checksum_address
```

```
if is_checksum_address(address):  
    return address
```

```
return to_checksum_address(address)
```

```
except Exception:  
    return None
```

```
@classmethod  
def validate_tx_hash(cls, tx_hash: str) -> Optional[str]:  
    if not tx_hash:  
        return None
```

```
tx_hash = tx_hash.strip().lower()
```

```
if not cls.TX_HASH_PATTERN.match(tx_hash):  
    return None
```

```
return tx_hash
```

```
@classmethod  
def validate_amount(cls, amount_str: str, min_val=0,  
max_val=None) -> Optional[float]:  
    if not amount_str:  
        return None
```

```
amount_str = amount_str.strip()
```

```
if not cls.AMOUNT_PATTERN.match(amount_str):
```

```
return None
```

```
try:
```

```
    amount = float(amount_str)
```

```
    if amount < min_val:
```

```
        return None
```

```
    if max_val and amount > max_val:
```

```
        return None
```

```
    return amount
```

```
except ValueError:
```

```
    return None
```

```
@classmethod
```

```
def validate_percentage(cls, value: str) -> Optional[float]:
```

```
    amount = cls.validate_amount(value, min_val=0, max_val=100)
```

```
    return amount
```

```
@classmethod
```

```
def sanitize_text(cls, text: str, max_length=1000) -> str:
```

```
    if not text:
```

```
        return ""
```

```
    text = text[:max_length]
```

```
    dangerous_patterns = [
```

```
        (r"<script.*?>.*?</script>", ""),
```

```
        (r"javascript:", ""),
```

```
        (r"on\w+\s*=", ""),
```

```
    ]
```

```
    for pattern, replacement in dangerous_patterns:
```

```
        text = re.sub(pattern, replacement, text, flags=re.IGNORECASE)
```

```
    return text
```

```
@classmethod
```



```

def validate_telegram_user_id(cls, user_id) -> Optional[int]:
    try:
        uid = int(user_id)

        if uid <= 0:
            return None

        if uid > 100000000000:
            return None

        return uid

    except (ValueError, TypeError):
        return None

```

Command Argument Parser:

```

class CommandParser:

```

```

    def __init__(self):
        self.validator = InputValidator()

    def parse_buy_command(self, args):
        if len(args) < 2:
            return None, "Usage: /buy [token] [amount]"

```

```

        token = self.validator.validate_eth_address(args[0])
        if not token:
            return None, "Invalid token address"

```

```

        amount = self.validator.validate_amount(args[1],
        min_val=0.0001, max_val=100)
        if not amount:
            return None, "Invalid amount (0.0001 - 100)"

```

```

        slippage = 0.5
        if len(args) > 2:
            slippage = self.validator.validate_amount(args[2], min_val=0.1,
            max_val=50)

```

```
if not slippage:  
    return None, "Invalid slippage (0.1 - 50)"
```

```
    return {  
        "token": token,  
        "amount": amount,  
        "slippage": slippage  
    }, None
```

```
def parse_sell_command(self, args):  
    if len(args) < 2:  
        return None, "Usage: /sell [token] [amount/%]"
```

```
    token = self.validator.validate_eth_address(args[0])  
    if not token:  
        return None, "Invalid token address"
```

```
    amount_str = args[1].strip()
```

```
    if amount_str.endswith("%"):  
        percentage = self.validator.validate_percentage(amount_str[:-1])  
        if not percentage:  
            return None, "Invalid percentage (1-100)"
```

```
    return {  
        "token": token,  
        "percentage": percentage,  
        "amount": None  
    }, None
```

```
    else:  
        amount = self.validator.validate_amount(amount_str, min_val=1)  
        if not amount:  
            return None, "Invalid amount"
```

```
    return {  
        "token": token,  
        "percentage": None,  
        "amount": int(amount)  
    }, None
```

SECTION 3: RATE LIMITING

Rate limits protect against abuse. User limits. API limits. Resource limits.

Rate Limiter:

```
import time
from collections import defaultdict
import asyncio
```

```
class RateLimiter:
```

```
    def __init__(self):
        self.user_requests = defaultdict(list)
        self.ip_requests = defaultdict(list)
        self.global_requests = []
```

```
    self.limits = {
        "user_per_minute": 30,
        "user_per_hour": 200,
        "ip_per_minute": 60,
        "global_per_second": 100
    }
```

```
    async def check_rate_limit(self, user_id, ip=None):
        now = time.time()
```

```
        self._cleanup(now)
```

```
        user_minute = self._count_recent(self.user_requests[user_id], now,
60)
        if user_minute >= self.limits["user_per_minute"]:
            return False, "Rate limit exceeded. Try again in a minute."
```

```
        user_hour = self._count_recent(self.user_requests[user_id], now,
3600)
        if user_hour >= self.limits["user_per_hour"]:
```

```
return False, "Hourly rate limit exceeded."
```

```
if ip:
```

```
    ip_minute = self._count_recent(self.ip_requests[ip], now, 60)
```

```
    if ip_minute >= self.limits["ip_per_minute"]:
```

```
        return False, "IP rate limit exceeded."
```

```
global_second = self._count_recent(self.global_requests, now, 1)
```

```
if global_second >= self.limits["global_per_second"]:
```

```
    return False, "System is busy. Try again shortly."
```

```
self.user_requests[user_id].append(now)
```

```
if ip:
```

```
    self.ip_requests[ip].append(now)
```

```
self.global_requests.append(now)
```

```
return True, None
```

```
def _count_recent(self, timestamps, now, window):
```

```
    cutoff = now - window
```

```
    return sum(1 for t in timestamps if t > cutoff)
```

```
def _cleanup(self, now):
```

```
    cutoff = now - 3600
```

```
for user_id in list(self.user_requests.keys()):
```

```
    self.user_requests[user_id] = [
```

```
        t for t in self.user_requests[user_id] if t > cutoff
```

```
    ]
```

```
if not self.user_requests[user_id]:
```

```
    del self.user_requests[user_id]
```

```
for ip in list(self.ip_requests.keys()):
```

```
    self.ip_requests[ip] = [
```

```
        t for t in self.ip_requests[ip] if t > cutoff
```

```
    ]
```

```
if not self.ip_requests[ip]:
```

```
    del self.ip_requests[ip]
```

```
self.global_requests = [t for t in self.global_requests if t > now -
```

10]

```
class OperationRateLimiter:
```

```
    def __init__(self):
```

```
        self.operation_counts = defaultdict(lambda: defaultdict(int))
```

```
        self.last_reset = defaultdict(float)
```

```
        self.operation_limits = {
```

```
            "trade": {"per_minute": 10, "per_day": 100},
```

```
            "withdraw": {"per_hour": 3, "per_day": 10},
```

```
            "copy_trade": {"per_minute": 20, "per_day": 500}
```

```
        }
```

```
    async def check_operation_limit(self, user_id, operation):
```

```
        now = time.time()
```

```
        self._maybe_reset(user_id, operation, now)
```

```
        limits = self.operation_limits.get(operation)
```

```
        if not limits:
```

```
            return True, None
```

```
        count = self.operation_counts[user_id][operation]
```

```
        per_minute = limits.get("per_minute", float("inf"))
```

```
        if count >= per_minute:
```

```
            return False, f"Too many {operation} operations. Limit:  
{per_minute}/minute"
```

```
        self.operation_counts[user_id][operation] += 1
```

```
        return True, None
```

```
    def _maybe_reset(self, user_id, operation, now):
```

```
        key = f"{user_id}:{operation}"
```

```
        last = self.last_reset.get(key, 0)
```

```
        if now - last > 60:
```

```
            self.operation_counts[user_id][operation] = 0
```

```
self.last_reset[key] = now
```

SECTION 4: SECURE COMMUNICATION

All communication must be secure. Encrypt sensitive data. Verify authenticity.

Secure Telegram Communication:

```
import hmac
import hashlib

class SecureTelegram:

    def __init__(self, bot_token):
        self.bot_token = bot_token
        self.secret_key = hashlib.sha256(bot_token.encode()).digest()

    def verify_callback_data(self, data, signature):
        expected = hmac.new(
            self.secret_key,
            data.encode(),
            hashlib.sha256
        ).hexdigest()

        return hmac.compare_digest(expected, signature)

    def sign_callback_data(self, data):
        signature = hmac.new(
            self.secret_key,
            data.encode(),
            hashlib.sha256
        ).hexdigest()

        return f'{data}:{signature[:16]}'

    def verify_signed_callback(self, signed_data):
        if ":" not in signed_data:
            return None
```

```
data, signature = signed_data.rsplit(":", 1)

expected = hmac.new(
    self.secret_key,
    data.encode(),
    hashlib.sha256
).hexdigest()[:16]

if hmac.compare_digest(expected, signature):
    return data

return None
```

Webhook Verification:

```
class WebhookVerifier:

    def __init__(self, bot_token):
        self.secret_token = hashlib.sha256(
            f'webhook:{bot_token}'.encode()
        ).hexdigest()[:32]

    def get_secret_token(self):
        return self.secret_token

    def verify_request(self, request_token):
        if not request_token:
            return False

        return hmac.compare_digest(self.secret_token, request_token)
```

Encrypted Storage:

```
class EncryptedStorage:

    def __init__(self, encryption_key):
        from cryptography.fernet import Fernet
```

```
if len(encryption_key) != 32:
    key_bytes = hashlib.sha256(encryption_key.encode()).digest()
    self.cipher = Fernet(base64.urlsafe_b64encode(key_bytes))
else:
    self.cipher =
Fernet(base64.urlsafe_b64encode(encryption_key.encode()))
```

```
def encrypt(self, data):
    if isinstance(data, str):
        data = data.encode()
    return self.cipher.encrypt(data).decode()
```

```
def decrypt(self, encrypted_data):
    if isinstance(encrypted_data, str):
        encrypted_data = encrypted_data.encode()
    return self.cipher.decrypt(encrypted_data).decode()
```

```
def encrypt_json(self, data):
    import json
    json_str = json.dumps(data)
    return self.encrypt(json_str)
```

```
def decrypt_json(self, encrypted_data):
    import json
    json_str = self.decrypt(encrypted_data)
    return json.loads(json_str)
```

SECTION 5: ERROR HANDLING

Errors reveal information. Handle them carefully. Never expose internal details.

Secure Error Handler:

```
import traceback
import uuid
```

```
class SecureErrorHandler:
```



```

def __init__(self, logger):
    self.logger = logger

    async def handle_error(self, error, context=None):
        error_id = str(uuid.uuid4())[0:8]

        self.logger.error(
            f'Error {error_id}: {type(error).__name__}: {str(error)}',
            extra={
                "error_id": error_id,
                "error_type": type(error).__name__,
                "traceback": traceback.format_exc(),
                "context": context
            }
        )

        user_message = self.get_user_message(error, error_id)

        return user_message, error_id

    def get_user_message(self, error, error_id):
        error_type = type(error).__name__

        if error_type in ["ValueError", "ValidationError"]:
            return f"Invalid input. Please check your command."

        if error_type == "InsufficientFundsError":
            return "Insufficient balance for this operation."

        if error_type == "RateLimitError":
            return "Too many requests. Please wait and try again."

        if error_type == "NetworkError":
            return "Network issue. Please try again shortly."

        return f"An error occurred. Reference: {error_id}"

```

```

class ErrorMiddleware:

```

```

def _init_(self, error_handler):
    self.handler = error_handler

def wrap_handler(self, func):
    async def wrapped(update, context):
        try:
            return await func(update, context)

        except Exception as e:
            user_message, error_id = await self.handler.handle_error(
                e,
                context = {
                    "user_id": update.effective_user.id if update.effective_user else None,
                    "chat_id": update.effective_chat.id if update.effective_chat else
                    None,
                    "command": update.message.text if update.message else None
                }
            )

            if update.message:
                await update.message.reply_text(user_message)
            elif update.callback_query:
                await update.callback_query.answer(user_message,
                    show_alert = True)

            return wrapped

```

SECTION 6: AUDIT LOGGING

Every significant action must be logged. Immutable audit trail.
Compliance ready.

Audit Logger:

```

import json
from datetime import datetime

class AuditLogger:

```

```
def __init__(self, database):
    self.db = database
```

```
    async def log_action(self, action_type, user_id, details,
ip_address=None):
        log_entry = {
            "timestamp": datetime.utcnow().isoformat(),
            "action": action_type,
            "user_id": user_id,
            "details": details,
            "ip_address": ip_address
        }
```

```
        await self.db.execute("""
INSERT INTO audit_log
(action_type, user_id, details, ip_address, created_at)
VALUES ($1, $2, $3, $4, NOW())
""", action_type, user_id, json.dumps(details), ip_address)
```

```
    return log_entry
```

```
    async def log_trade(self, user_id, trade_details):
        await self.log_action(
            "trade",
            user_id,
            {
                "type": trade_details.get("type"),
                "token": trade_details.get("token"),
                "amount": str(trade_details.get("amount")),
                "tx_hash": trade_details.get("tx_hash")
            }
        )
```

```
    async def log_withdrawal(self, user_id, withdrawal_details):
        await self.log_action(
            "withdrawal",
            user_id,
            {
                "amount": str(withdrawal_details.get("amount")),
```

```
"destination": withdrawal_details.get("destination"),
"tx_hash": withdrawal_details.get("tx_hash")
}
)
```

```
async def log_login(self, user_id, login_details):
    await self.log_action(
        "login",
        user_id,
        {
            "platform": login_details.get("platform"),
            "success": login_details.get("success")
        },
        ip_address = login_details.get("ip")
    )
```

```
async def log_settings_change(self, user_id, old_settings,
new_settings):
    changes = {}
```

```
    for key in set(list(old_settings.keys()) + list(new_settings.keys())):
        old_val = old_settings.get(key)
        new_val = new_settings.get(key)
```

```
    if old_val != new_val:
        changes[key] = {"old": old_val, "new": new_val}
```

```
    if changes:
        await self.log_action(
            "settings_change",
            user_id,
            {"changes": changes}
        )
```

```
async def get_user_history(self, user_id, action_type = None,
limit = 100):
    if action_type:
        rows = await self.db.fetch("""
        SELECT * FROM audit_log
        WHERE user_id = $1 AND action_type = $2
```

```

ORDER BY created_at DESC
LIMIT $3
"""', user_id, action_type, limit)
else:
rows = await self.db.fetch("""
SELECT * FROM audit_log
WHERE user_id = $1
ORDER BY created_at DESC
LIMIT $2
"""', user_id, limit)

return [dict(row) for row in rows]

```

Audit Log Schema:

```

audit_schema = """

CREATE TABLE audit_log (
  id BIGSERIAL PRIMARY KEY,
  action_type VARCHAR(50) NOT NULL,
  user_id BIGINT NOT NULL,
  details JSONB NOT NULL,
  ip_address INET,
  created_at TIMESTAMP DEFAULT NOW()
);

CREATE INDEX idx_audit_user ON audit_log(user_id);
CREATE INDEX idx_audit_type ON audit_log(action_type);
CREATE INDEX idx_audit_created ON audit_log(created_at);

ALTER TABLE audit_log SET (
  autovacuum_enabled = true,
  toast.autovacuum_enabled = true
);

"""

```

SECTION 7: USER FUND PROTECTION

Protecting user funds is the highest priority. Multiple safeguards.

Fund Protection System:

```
class FundProtection:
```

```
    def __init__(self, database, notification_service):
        self.db = database
        self.notify = notification_service
```

```
    self.daily_withdrawal_limit = 10
    self.single_withdrawal_limit = 5
    self.suspicious_threshold = 0.5
```

```
    async def validate_withdrawal(self, user_id, amount, destination):
        checks = [
            self.check_daily_limit(user_id, amount),
            self.check_single_limit(amount),
            self.check_destination(destination),
            self.check_unusual_activity(user_id, amount)
        ]
```

```
        for check in checks:
            result = await check
```

```
        if not result["approved"]:
            await self.notify.alert_security(
                "withdrawal_blocked",
                {
                    "user_id": user_id,
                    "amount": str(amount),
                    "reason": result["reason"]
                }
            )
        return result
```

```
    return {"approved": True}
```

```
    async def check_daily_limit(self, user_id, amount):
```

```

daily_total = await self.db.fetchval("""
SELECT COALESCE(SUM((details->>'amount')::NUMERIC), 0)
FROM audit_log
WHERE user_id = $1
AND action_type = 'withdrawal'
AND created_at > NOW() - INTERVAL '24 hours'
""", user_id)

```

```

if daily_total + amount > self.daily_withdrawal_limit:
    return {
        "approved": False,
        "reason": f"Daily withdrawal limit ({self.daily_withdrawal_limit}
ETH) exceeded"
    }

```

```

return {"approved": True}

```

```

async def check_single_limit(self, amount):
    if amount > self.single_withdrawal_limit:
        return {
            "approved": False,
            "reason": f"Single withdrawal limit ({self.single_withdrawal_limit}
ETH) exceeded"
        }

```

```

return {"approved": True}

```

```

async def check_destination(self, destination):
    is_blacklisted = await self.db.fetchval("""
SELECT EXISTS(
SELECT 1 FROM blacklisted_addresses
WHERE address = $1
)
""", destination.lower())

```

```

if is_blacklisted:
    return {
        "approved": False,
        "reason": "Destination address is blacklisted"
    }

```

```
return {"approved": True}
```

```
async def check_unusual_activity(self, user_id, amount):  
    user_balance = await self.db.fetchval("""  
    SELECT balance FROM user_balances WHERE user_id = $1  
    """, user_id)
```

```
    if not user_balance:  
        return {"approved": True}
```

```
    if amount / user_balance > self.suspicious_threshold:  
        await self.require_confirmation(user_id, amount)  
        return {  
            "approved": False,  
            "reason": "Large withdrawal requires confirmation",  
            "requires_confirmation": True  
        }
```

```
    return {"approved": True}
```

```
async def require_confirmation(self, user_id, amount):  
    confirmation_code = self.generate_confirmation_code()
```

```
    await self.db.execute("""  
    INSERT INTO pending_confirmations  
    (user_id, action_type, details, code, expires_at)  
    VALUES ($1, 'withdrawal', $2, $3, NOW() + INTERVAL '10  
    minutes')  
    """, user_id, {"amount": str(amount)}, confirmation_code)
```

```
    await self.notify.send_confirmation_request(user_id,  
    confirmation_code)
```

```
def generate_confirmation_code(self):  
    import secrets  
    return secrets.token_hex(3).upper()
```

Hot and Cold Wallet Management:


```

class WalletManager:

    def __init__(self, config):
        self.hot_wallet_threshold = config.hot_wallet_threshold
        self.hot_wallet = config.hot_wallet_address

    async def process_deposit(self, user_id, amount, tx_hash):
        if amount > self.hot_wallet_threshold:
            await self.transfer_to_cold(amount - self.hot_wallet_threshold)

    async def process_withdrawal(self, user_id, amount, destination):
        hot_balance = await self.get_hot_balance()

        if amount > hot_balance:
            await self.request_cold_refill(amount - hot_balance)
            return {
                "status": "pending",
                "reason": "Processing. Large withdrawal requires manual review."
            }

        return await self.execute_withdrawal(amount, destination)

    async def get_hot_balance(self):
        pass

    async def transfer_to_cold(self, amount):
        pass

    async def request_cold_refill(self, amount):
        pass

    async def execute_withdrawal(self, amount, destination):
        pass

```

SECTION 8: AUTHENTICATION AND AUTHORIZATION

Verify user identity. Control access to features.

Authentication System:

```
import time
import secrets
```

```
class AuthenticationSystem:
```

```
    def __init__(self, database):
        self.db = database
        self.session_timeout = 86400
```

```
    async def create_session(self, user_id, telegram_id):
        session_token = secrets.token_urlsafe(32)
```

```
        await self.db.execute("""
INSERT INTO user_sessions
(user_id, telegram_id, session_token, created_at, expires_at)
VALUES ($1, $2, $3, NOW(), NOW() + INTERVAL '24 hours')
""", user_id, telegram_id, session_token)
```

```
        return session_token
```

```
    async def verify_session(self, session_token):
        session = await self.db.fetchrow("""
SELECT * FROM user_sessions
WHERE session_token = $1
AND expires_at > NOW()
""", session_token)
```

```
        if not session:
            return None
```

```
        await self.db.execute("""
UPDATE user_sessions
SET last_used = NOW()
WHERE session_token = $1
""", session_token)
```

```
        return session["user_id"]
```

```
async def invalidate_session(self, session_token):
    await self.db.execute("""
DELETE FROM user_sessions
WHERE session_token = $1
""", session_token)
```

```
async def invalidate_all_sessions(self, user_id):
    await self.db.execute("""
DELETE FROM user_sessions
WHERE user_id = $1
""", user_id)
```

Authorization System:

```
class AuthorizationSystem:
```

```
    def __init__(self, database):
        self.db = database
```

```
        self.permissions = {
            "basic": ["view_balance", "view_positions", "trade"],
            "premium": ["view_balance", "view_positions", "trade", "snipe",
"copy_trade"],
            "admin": ["*"]
        }
```

```
    async def get_user_role(self, user_id):
        role = await self.db.fetchval("""
SELECT role FROM users WHERE id = $1
""", user_id)
```

```
        return role or "basic"
```

```
    async def has_permission(self, user_id, permission):
        role = await self.get_user_role(user_id)
```

```
        role_permissions = self.permissions.get(role, [])
```

```
        if "*" in role_permissions:
```

```
return True
```

```
return permission in role_permissions
```

```
def require_permission(self, permission):
```

```
def decorator(func):
```

```
    async def wrapper(update, context):
```

```
        user_id = update.effective_user.id
```

```
        if not await self.has_permission(user_id, permission):
```

```
            await update.message.reply_text(
```

```
                "You don't have permission for this action."
```

```
            )
```

```
        return
```

```
    return await func(update, context)
```

```
    return wrapper
```

```
    return decorator
```

SECTION 9: SECURITY MONITORING

Detect threats in real-time. Alert on suspicious activity.

Security Monitor:

```
from collections import defaultdict
```

```
import asyncio
```

```
class SecurityMonitor:
```

```
    def __init__(self, notification_service):
```

```
        self.notify = notification_service
```

```
        self.failed_attempts = defaultdict(list)
```

```
        self.suspicious_activity = defaultdict(list)
```

```
        self.running = False
```

```
async def start(self):
    self.running = True
    asyncio.create_task(self.cleanup_loop())
```

```
async def cleanup_loop(self):
    while self.running:
        await asyncio.sleep(300)
```

```
now = time.time()
cutoff = now - 3600
```

```
for key in list(self.failed_attempts.keys()):
    self.failed_attempts[key] = [
        t for t in self.failed_attempts[key] if t > cutoff
    ]
```

```
async def record_failed_attempt(self, user_id, action):
    now = time.time()
    self.failed_attempts[user_id].append(now)
```

```
recent_failures = len([
    t for t in self.failed_attempts[user_id]
    if t > now - 300
])
```

```
if recent_failures >= 5:
    await self.notify.alert_security(
        "multiple_failed_attempts",
        {
            "user_id": user_id,
            "action": action,
            "count": recent_failures
        }
    )
```

```
async def detect_anomaly(self, user_id, activity_type, details):
    anomalies = []
```

```
if activity_type == "trade":
    if details.get("amount", 0) > details.get("usual_amount", 0) * 10:
```

```
anomalies.append("unusual_trade_size")
```

```
if activity_type == "login":  
    if details.get("new_device"):  
        anomalies.append("new_device")
```

```
if details.get("unusual_location"):  
    anomalies.append("unusual_location")
```

```
if anomalies:  
    await self.notify.alert_security(  
        "anomaly_detected",  
        {  
            "user_id": user_id,  
            "activity": activity_type,  
            "anomalies": anomalies,  
            "details": details  
        }  
    )
```

```
return anomalies
```

Notification Service:

```
class SecurityNotificationService:
```

```
    def __init__(self, config):  
        self.admin_chat_ids = config.admin_chat_ids  
        self.bot_token = config.bot_token
```

```
    async def alert_security(self, alert_type, details):  
        message = self.format_alert(alert_type, details)
```

```
        for admin_id in self.admin_chat_ids:  
            await self.send_telegram_message(admin_id, message)
```

```
    def format_alert(self, alert_type, details):  
        templates = {  
            "withdrawal_blocked": (
```

```

"<b>WITHDRAWAL BLOCKED</b>\n\n"
"User: {user_id}\n"
"Amount: {amount}\n"
"Reason: {reason}"
),
"multiple_failed_attempts": (
"<b>SUSPICIOUS ACTIVITY</b>\n\n"
"User: {user_id}\n"
"Action: {action}\n"
"Failed attempts: {count}"
),
"anomaly_detected": (
"<b>ANOMALY DETECTED</b>\n\n"
"User: {user_id}\n"
"Activity: {activity}\n"
"Anomalies: {anomalies}"
)
}

```

```

template = templates.get(alert_type, str(details))
return template.format(**details)

```

```

async def send_telegram_message(self, chat_id, message):
import aiohttp

```

```

url = f"https://api.telegram.org/bot{self.bot_token}/sendMessage"

```

```

async with aiohttp.ClientSession() as session:
await session.post(url, json={
"chat_id": chat_id,
"text": message,
"parse_mode": "HTML"
})

```

```

async def send_confirmation_request(self, user_id, code):
message = (
"<b>Confirmation Required</b>\n\n"
f"A large withdrawal was requested.\n"
f"Confirmation code: <code>{code}</code>\n\n"
"Reply with /confirm [code] to proceed.\n"

```

```
"This code expires in 10 minutes."
```

```
)
```

```
await self.send_telegram_message(user_id, message)
```

SECTION 10: PRODUCTION DEPLOYMENT

Final deployment checklist and best practices.

Deployment Checklist:

```
class DeploymentChecklist:
```

```
    def __init__(self):
```

```
        self.checks = {
```

```
            "security": [
```

```
                "All secrets in environment variables",
```

```
                "Private keys encrypted",
```

```
                "API keys have minimum required permissions",
```

```
                "Webhook uses HTTPS only",
```

```
                "Input validation on all user inputs",
```

```
                "Rate limiting configured",
```

```
                "Audit logging enabled",
```

```
                "Error messages don't expose internals"
```

```
            ],
```

```
            "reliability": [
```

```
                "Multiple RPC providers configured",
```

```
                "Database connection pooling",
```

```
                "Graceful shutdown handling",
```

```
                "Health check endpoint",
```

```
                "Automatic restart on failure",
```

```
                "Database backups scheduled",
```

```
                "Log rotation configured"
```

```
            ],
```

```
            "monitoring": [
```

```
                "Error alerting configured",
```

```
                "Performance metrics collected",
```

```
                "Uptime monitoring",
```

```
                "Resource usage alerts",
```



```

"Security event alerts"
],
"compliance": [
"Terms of service published",
"Privacy policy published",
"Risk disclaimers visible",
"Data retention policy implemented",
"User data export capability"
]
}

```

```

def run_checks(self, system):
    results = {}

    for category, items in self.checks.items():
        results[category] = []

        for item in items:
            passed = self.verify_check(system, item)
            results[category].append({
                "check": item,
                "passed": passed
            })

    return results

def verify_check(self, system, check):
    return True

```

Docker Configuration:

```
dockerfile_content = """
```

```
FROM python:3.11-slim
```

```
WORKDIR /app
```

```

RUN apt-get update && apt-get install -y --no-install-recommends \\\
gcc \\\

```

```
libpq-dev \\  
&& rm -rf /var/lib/apt/lists/*
```

```
COPY requirements.txt .  
RUN pip install --no-cache-dir -r requirements.txt
```

```
COPY . .
```

```
RUN useradd -m -u 1000 botuser && chown -R botuser:botuser /  
app  
USER botuser
```

```
CMD ["python", "-m", "bot.main"]
```

```
"""
```

```
docker_compose_content = """
```

```
version: '3.8'
```

```
services:
```

```
  bot:
```

```
    build: .
```

```
    restart: unless-stopped
```

```
    environment:
```

```
      - BOT_TOKEN=${BOT_TOKEN}
```

```
      - DATABASE_URL=${DATABASE_URL}
```

```
      - ENCRYPTION_KEY=${ENCRYPTION_KEY}
```

```
      - RPC_URL=${RPC_URL}
```

```
    depends_on:
```

```
      - postgres
```

```
      - redis
```

```
    networks:
```

```
      - bot-network
```

```
postgres:
```

```
  image: postgres:15
```

```
  restart: unless-stopped
```

```
  environment:
```

```
    - POSTGRES_DB=botdb
```

```
- POSTGRES_USER = ${DB_USER}
- POSTGRES_PASSWORD = ${DB_PASSWORD}
volumes:
- postgres-data:/var/lib/postgresql/data
networks:
- bot-network
```

```
redis:
image: redis:7-alpine
restart: unless-stopped
networks:
- bot-network
```

```
networks:
bot-network:
```

```
volumes:
postgres-data:
```

```
""""
```

Health Check Endpoint:

```
from aiohttp import web
```

```
class HealthCheck:
```

```
def __init__(self, bot):
self.bot = bot
```

```
async def check_database(self):
try:
await self.bot.db.fetchval("SELECT 1")
return True
except Exception:
return False
```

```
async def check_telegram(self):
try:
```

```

await self.bot.application.bot.get_me()
return True
except Exception:
return False

async def health_endpoint(self, request):
db_ok = await self.check_database()
telegram_ok = await self.check_telegram()

status = "healthy" if (db_ok and telegram_ok) else "unhealthy"

return web.json_response({
    "status": status,
    "checks": {
        "database": "ok" if db_ok else "error",
        "telegram": "ok" if telegram_ok else "error"
    }
}, status=200 if status == "healthy" else 503)

def create_app(self):
app = web.Application()
app.router.add_get("/health", self.health_endpoint)
return app

```

CHAPTER SUMMARY

This chapter covered security and best practices:

Private key security uses encryption at rest, hardware security modules for production, and environment variables for secrets.

Input validation sanitizes all user input. Ethereum addresses, amounts, text - everything is validated before use.

Rate limiting protects against abuse at user, operation, and system levels.

Secure communication verifies webhook authenticity, signs callback data, and encrypts sensitive storage.

Error handling prevents information leakage while logging full details for debugging.

Audit logging creates immutable records of all significant actions for compliance and investigation.

Fund protection implements withdrawal limits, destination verification, and unusual activity detection.

Authentication and authorization verify identity and control feature access based on roles.

Security monitoring detects threats in real-time and alerts administrators.

Production deployment requires security, reliability, monitoring, and compliance checklists.

BOOK 5 SUMMARY

This book covered Telegram bot integration:

Chapter 1: Telegram Platform Overview - Architecture, API types, rate limits.

Chapter 2: Bot Development Fundamentals - Commands, handlers, keyboards, state management.

Chapter 3: Mini Apps Development - HTML5 apps inside Telegram, initialization, payments.

Chapter 4: Telegram Stars Payments - Virtual currency, payment flows, refunds.

Chapter 5: TON Blockchain Integration - TON Connect, wallet connection, transactions.

Chapter 6: Trading Bot Architecture - Order management, sniping,

copy trading, MEV protection.

Chapter 7: Community Management - Welcome flows, anti-spam, moderation, engagement tracking.

Chapter 8: Security Best Practices - Key management, validation, rate limiting, fund protection.

Next book: Advanced DEX Development. We build decentralized exchanges.